Mathematica® programming: an advanced introduction

Leonid Shifrin

Part I: The core language

Version 1.01

Mathematica programming: an advanced introduction

Leonid Shifrin

To my parents

Contents

List of case studies and selected examples

Examples

Preface

The history of this project

I started using *Mathematica* about 10 years ago for my Masters thesis. Since then, I have been using it occasionally during my PhD, until about 3 years ago. At that point, just for curiosity, I tried to use *Mathematica* for a small side project which had nothing to do with the field of my professional activity (theoretical physics). And then, I have suddenly discovered that behind all the built - in commands and simple procedural programming constructs there is a much more powerful programming language (the fact of course well-known by then to lots of people, but not to me). I was hooked and spent some time experimenting with its different features and then read a few books to get a deeper insight into it. At that time, it was mostly for my own amusement, since ways in which I used *Mathematica* professionally did not require even a fraction of the full power of this language. However, the character of my work has changed since, and it’s been about one and a half years now that I use *Mathematica* heavily on a daily basis and very frequently need the full power it can give me, in terms of speed, numerical and symbolic capabilities. And I can safely say that without the knowledge of how to program in it properly, most of my recent scientific results would be a lot harder to get. At some point, I decided to create some notes on *Mathematica* programming, mainly for myself, and also to somehow organize the code that I have accumulated so far for various projects. But as the notes started to expand, it occurred to me that with some more effort I could convert them into a text possibly useful for other people. So, that’s how it started.

The audience for this book

When writing this book I mostly had in mind people who want to understand *Mathematica* programming, and particularly those *Mathematica* users who would like to make a transition from a user to a programmer, or perhaps those who already have some limited *Mathematica* programming experience but want to improve their command of the system. Expert *Mathematica* programmers will probably find little new information in the book - may be, the last chapter could be of some interest to them.

The first part of the audience for this book are scientists who would like to understand *Mathematica* programming better, to take advantage of the possibilities it offers. The second part are (software) engineers who may consider *Mathematica* as a tool for a prototype design. In this context, *Mathematica* can serve as a tool of "experimental programming", especially useful in projects where some non-trivial computations/research have to accompany programming.

■ Why *Mathematica*?

At the end of the day, there is nothing that can be done in *Mathematica* and absolutely can not be done in other programming environments. For many problems however, especially those involving symbolic programming, solving a problem in a language such as C or C++ will be eventually equivalent to reimplementing a subset of *Mathematica* (or other system for symbolic manipulations) needed to solve the problem. The point is that many things are done in *Mathematica* with less or a lot less effort and time, because a lot of both generic and specific functionality is already built in *Mathematica*. And because it is so general, I expect this statement to be true for almost any field where some computations, prototype or program design and development, simulations etc are used. *Mathematica* seems to be an ideal tool for development of toy - models, prototypes, or just ideas. While *Mathematica* may be also quite useful for validating some ideas or solutions, as well as to power some quite complex technologies also in their final form, my feeling is that it may be most useful as a tool of experimental research (or programming), where the answer (or design) is not known in advance.

For the scientific part of my audience, it is probably easier to argue in favour of *Mathematica*, since the end product in science is usually a solution of certain problem, and *Mathematica* serves as a tool of research. Its value here is that it has many built-in functions and commands which allow to do a lot of things quickly.

On the other hand, there are many great programming languages, environments and tools. Many of them have an added advantage of being free and open source. For the programming and prototype design purposes, one may well question the advantages of using a proprietary software, which also is intrinsically built in a way that does not allow to make an executable directly (it would require to package the entire kernel together with your code and lead to a very large size of an executable. The *Mathematica* Player technology seems to be a step in this direction).

Here are 10 good reasons to use *Mathematica*:

1. *Multiparadigm language*: the richness of the language allows to pick for any problem a programming paradigm or style which corresponds to it most directly. You spend most of the time thinking about the problem rather than implementation. The very high level of the language means that a lot of work is done for you by the system.

2. *Interactivity*. *Mathematica* is an interpreted language, which allows interactive and incremental program development. The *Mathematica* front - end adds another layer of interactivity, being able to display various forms of input and output (and this can be controlled programmatically). Yet another layer of interactivity is added by many new features of version 6.

3. *Programming in the large*. The typically small size and high level of abstraction of the code allows a single person to manage substantial projects. There is also a built-in support for large projects through the system of packages.

3. *Built-ins*. Availability of thousands of built-in functions makes it possible to do sophisticated analysis very quickly. Extended error message system (each built-in function can issue a lot of error messages on improper inputs) greatly simplifies debugging.

4. *Genericity, higher-order functions and tight system integration*. The very general principles of *Mathematica*, its uniform expression structure, generic nature of many built-in functions, and tight integration of

all components allows to use all other built-in functions much easier than one would use libraries in other languages. The Help system is also uniform and it is immediate to learn the functionality of any built-in function that you have never used before.

6. *Visualizations*. Great dynamic and visualization capabilities (especially in version 6).

7. *Cross-platform*. The *Mathematica* code developed in one environment or OS will work in exactly the same way in all others where *Mathematica* is available.

8. *Connectivity*: the developers keep increasing the number of file formats which *Mathematica* can understand and process. Also, tools like *MathLink* , *J/Link* , database connectivity etc. allow one to connect *Mathematica* to external programs

9. *Backward compatibility*: since the version 1 and up to these days developers are careful to maintain very high level of backwards compatibility. This means that one should not worry too much that solutions developed in the current version will need a rewrite to work on the future versions (apart from possible improvements related to availability of new built - in functions, if one is so inclined).

10. *Support for parallel and distributed computing*.

In addition to this, version 6 front - end contains a built - in mini - IDE (text highlighting which is aware of the syntax of built - in commands, allows to automatically track scope of variables, etc.; package creating and testing greatly simplified; interactive debugger). These features make version 6 a full-blown development environment - I personally found it much more fun to develop code in it than in the previous versions. Also, there is Eclipse - based Wolfram Workbench IDE for development of larger projects.

■ The choice of the material

Since the first part of the tutorial is devoted to the core language of *Mathematica* (or, if you wish, important built - in commands), the choice of material and even examples in this part are necessarily mostly "standard". This means that there will be a lot of overlaps with many existing sources, and many things are actually explained in much more detail elsewhere. I have included small discussions of some tidbits based on personal experience with certain specific cases, where I felt appropriate.

One more comment due here is that I made an emphasis on the functional subset of *Mathematica* language, which means that the book is geared more towards software development. The rule - based approach is covered but perhaps under - represented, which I hope to remedy in the next part (s) of this tutorial. Here I just want to emphasize that while the functional layer of *Mathematica* is nice for writing fast and compact programs, it is the rule-based engine that gives *Mathematica* real uniqueness, power and generality.

■ The style of presentation

I firmly believe that the best way to learn *Mathematica* programming is to learn it as a natural language. *Mathematica* programming language is very rich and in fact "overcomplete" in the sense that many built - in functions are in principle derivable from other built - in functions or their combinations. However, it is not an unstructured collection of functions "for everything" - it is built on powerful principles and has a uniform structure. To find a way through this large number of commands and understand their relative importance and relevance for a particular problem, it seems best to me to study the main principles and the structure of the language, but then go through the many language idioms and illustrate the use of each with many examples.

Thus, my way of presenting *Mathematica* programming can be characterized as language-driven and example-driven, but, unlike many other books, I do not cover separately different programming styles (procedural, rule-based, functional). Rather, I try to often give more than one implementation for a solution to a given problem, usually using different programming styles, and then discuss the differences, advantages and disadvantages of each on the level of individual examples. Because really, choosing your programming style before you start to understand the problem is like choosing tools to fix the car without knowing what’s broken. For the same reasons, I deliberately avoided discussions of any of thousands of the specialized tasks that *Mathematica* can perform, and instead considered it from a pure programming viewpoint. If we can imagine such a thing as "*Mathematica* cookbook", then I tried to make my book the exact opposite of it.

The examples I give are increasing in complexity as we go along, and in some cases I use constructs not yet covered, just for illustration (in such cases, it is best to concentrate on the part of the code which is currently discussed and ignore the unclear pieces, but revisit them later). However, many examples are admittedly rather contrived. This is because they serve to illustrate a given language idiom in the simplest possible setting. You will notice that many of the examples are concerned with imitation of the functionality of some built-in commands. This is not because I could not come up with more informative examples demonstrating the application of *Mathematica* to some "real world" problems, but because they are useful in understanding the system. By "reproducing" one built-in function with some combination of others, we not only learn about the inter-relations of various built-in commands, but also about performance wins and losses, avoiding the frustration associated with learning the same things "the hard way" on more complicated examples.

In my opinion, different programming techniques available in *Mathematica* in some sense split the language into several layers in terms of efficiency. I would call them scripting, intermediate and system layers. I try to introduce *Mathematica* programming in a way which at least does not completely ignore these language layers, by often providing alternative implementations for a given problem.

I hope to convince the reader that the advantages that *Mathematica* brings overweight the perhaps rather steep learning curve, and that *Mathematica* programming can be both useful and powerful, and a lot of fun.

Prerequisites

I assume that the reader has a basic familiarity with *Mathematica*, on the level of more or less the first part of the Stephen Wolfram' s *Mathematica* book [9]. In particular, while I discuss some parts of the syntax along the way, I do not systematically introduce the syntax from the ground up. However, I tried to make the book self - contained, and in principle it should be possible for someone completely new to *Mathematica* to follow the text, consulting *Mathematica* Help when things are unclear. Prior programming experience would be quite helpful (although not absolutely necessary) since I don' t discuss in a pedagogical manner the basic programming concepts such as loops etc.

The organization of the book

This first part of the tutorial is organized in 6 chapters.

Chapter 1 - *Introduction* - describes the main principles on which the *Mathematica* programming is based. I also have made a rather radical step to introduce along the way certain notions which are usually considered advanced and are discussed much later, such as DownValues or non - standard evaluation. But in my view, they are essential enough and simple enough to be at least touched early on.

Chapter 2 - *Elementary operations* - is mostly devoted to such absolutely essential elements of the *Mathematica* language as variables, assignments, equality checks etc. Here I also briefly describe the procedural control flow, branching, loops etc.

Chapter 3 - *Lists* - introduces lists as *Mathematica* main data structure building blocks, and then we go through many built-in functions available for various manipulations with lists. This chapter is rather large but quite important since it is essential to have a good handle on lists in *Mathematica* programming.

Chapter 4 - *Rules, patterns and functions* - has actually two major parts. The first one describes patterns and rules, and then the second one describes how one can define various functions in *Mathematica*. In fact, from the system point of view, rules and patterns are more fundamental than functions, the latter being just special kind of rules. I have combined them together for pragmatic reasons: people most commonly use the rule-based programming in *Mathematica* when they define their own functions. But frequently they have no idea about the role of rules and patterns in the functions they define, and this limits their possibilities or sometimes leads to misunderstandings and errors. Hopefully this chapter will clarify these issues.

Chapter 5 - *Functions on lists and functional programming* - is really the most important chapter in this part. It introduces functional programming - that is, application of functions to lists (data) and other functions. It builds up on the material of the previous two chapters. The notion of the higher order function is introduced, and then most of the important general - purpose higher - order functions are considered in detail and illustrated by many examples. Starting with this chapter, I also systematically emphasize performance considerations in *Mathematica* programming.

Chapter 6 - *Writing efficient programs* - the last chapter of this part, describes a few applications developed in the style discussed earlier. I present and compare several different implementations in each case. The main idea of this chapter is to show how a larger *Mathematica* program typically looks like, and which programming style is best for in which aspects. The case studies considered in this chapter can also serve as an illustration of several performance-tuning techniques.

which programming style is best for in which aspects. The case studies considered in this chapter can also serve as an illustration of several performance-tuning techniques.

There are also several appendices containing some additional information or remarks, and the bibliography.

Printing conventions

For the presentation of text, code, etc, I used the following conventions : each chapter, section, subsection and subsubsection are indexed. The maximum indexing depth in the book is 5 levels (the 5 - level index looks 1.2.3.4.5), although such long indexes are not always used. The names of chapters, sections, subsections and subsubsections are printed in Arial, the text is Times, example code is Courier Bold, and the output is Courier. The headers of some examples (typically smaller ones) is small Times Italic. The Times Italic Bold is used sometimes to separate logical steps in longer examples/case studies.

Some portions of the code are highlighted by a gray background. This is typically done to separate the logically more important pieces (like the code itself, a solution of some problem), from the other code (tests, checks, etc). Also, most complete functions are highlighted in this way.

How to use the book

The book can be either read systematically or one can just look at the topic of interest. Since it is generally example - based and centred around important built - in functions, inter - dependencies of different chapters or sections are generally not very strong, and mainly through the built - in functions used in the examples. Another use of the book could be as an additional source of examples of use for various built - in functions, and in this capacity it could supplement the standard examples from *Mathematica* Help or *Mathematica* Book. However, I did not have in mind to just assemble the collection of totally unrelated examples. So, for gaining a general understanding of the system there could be certain advantage in systematic reading of the book.

Another comment is that this book is no substitute for books containing more specific information on how to do certain types of mathematics, such as [10-12]. Neither is it a substitute for *Mathematica* manual, Help system or *Mathematica* Book [9]. Here, I stripped off all the aspects of *Mathematica* except those very closely related to its "programming engine", to make a pure programming book. But you will need also to know the other side of *Mathematica* to program real applications (although my experience is that this side is easy when you know how to program).

About the code in the book

Except when explicitly stated otherwise, all implementations are mine. I am the only one responsible for any errors present in the code (in the sense outlined in disclaimer below). I made an effort to ensure that it is reasonably bug-free. In particular, all examples were tested with *Mathematica* 5.2, and some with *Mathematica* 6.0. The code however was meant to serve purely pedagogical purposes, rather than to be any "production quality" (no extensive testing, arguments checks etc). It almost certainly does contain some bugs. If you find one (or more!), I will be most happy to learn about it, to get rid of it in the future versions of the book. If you decide to use the code for whatever purposes, however, do it at your own risk - please see the disclaimer.

Important topics not covered

Basically, I did not cover anything not related to the core language of *Mathematica*. This includes numerous functions computing integrals, derivatives, solving equations or inequalities of various types, plotting graphs etc - this material is covered in many texts, for instance [10-12]. But apart from these, several other important topics are missing. Perhaps the most serious omissions are: 1). I don't cover the wealth of new possibilities brought about by *Mathematica* 6. The partial excuse for this is that I focus on the core language which, as far as I could judge, did not change as much as some other features. 2). The *MathLink* protocol, and other connecting technologies like *J/Link* etc. 3). Certain topics such as front-end and notebook programming, or graphics and sound programming. 4). Working with the Wolfram WorkBench (the IDE) and using the debugger (version 6). 5) Internal compilation. 6) String operations, string-matching capabilities of *Mathematica,* regular expressions etc.

The main reason for these omissions is that I did not use these technologies as much in my work as to consider myself worthy of describing them. Besides, the volume of the book has grown way too much anyway, and these topics are still somewhat separate from the *Mathematica* language proper, which is the main focus of the book. Finally, some of these topics have received an excellent coverage elsewhere in the *Mathematica* literature.

■ License terms

■ The official web site

The official web site of the book is www.mathprogramming - intro.org [http://www.mathprogramming - intro.org]. You can download the latest version of the book from the web-site, and also send me a feedback. The online version of the book will also soon appear there.

■ Disclaimer

All the information in this book represents my strictly personal view on the *Mathematica*TM system. I am not affiliated with Wolfram Research Inc. in any way, as of the time of this writing.

All the information in this book is provided AS IS, with no warranty of any kind, expressed or implied. Neither I nor Wolfram Research Inc. will be liable, under any circumstances, in any loss of any form, direct or indirect, related to or caused by the use of any of the materials in this book.

Acknowledgements

First of all, my thanks go to the developers of *Mathematica* for the great system they have created. Apart from being a great aid in my research, using *Mathematica* changed my way of thinking, gave valuable insights and opened my eyes on many things which I probably would never think about otherwise. These thanks extend to authors of several excellent books on *Mathematica* which helped me understand the system much better, in particular Stephen Wolfram, David Wagner, Roman Maeder, John Gray and Michael Trott.

A number of people influenced my *Mathematica* - related activities in this or that way. I am grateful to Gernot Akemann for our fruitful collaboration which provided me with a source of countless problems that shaped my *Mathematica* skills, to Michael Phillips for the collaboration and nice discussions, and also for reassuring me of my sanity, to Alexey Pichugin for convincing me that C is a great language, to Jacques Bloch for collaboration and an opportunity to work together on a tough problem and learn about my limitations in *Mathematica*, to Dmitry Logunov for a joint project which did not quite work out but brought me down to earth, to PierPaolo Vivo for many nice discussions, a couple of nice problems to work on, and his ability to turn me into a good mood, and most importantly, to my wife Katya and daughter Anya for their patience and support during countless hours of my experimentation with *Mathematica* and then writing of this book.

Additionally, I am grateful to the members of the technical staff at SUNY at Stony Brook ,USA, and then at Brunel University West London, United Kingdom (particularly to Neil Turner) for providing me with an access to the most recent versions of *Mathematica*TM.

■ Comments, suggestions, criticism

I tried to make the book as self - contained and technically correct as I could. But please don't hesitate to contact me with any comments, suggestions or (preferably constructive) criticism - I will make all possible efforts to improve the book. You can e-mail me at

leonid@mathprogramming-intro.org

I. Introduction

Mathematica is built on a small number of universal principles. Good understanding of these principles is a pre-requisite for understanding how to program in *Mathematica*. Here I will discuss them, but rather briefly. Excellent and in-depth discussion of them can be found in several places [1,2,6 - 9].

■ 1.1 First principle: everything is an expression

The first principle states that every object dealt with by *Mathematica*, is an expression. Every *Mathematica* expression is either Atom, or a Normal Expression.

■ 1.1.1 Atoms and the built-in AtomQ predicate

Atoms are numbers, symbols and strings, and numbers are further divided into Integers, Reals, Rationals and Complex. All other objects are composite and are called Normal Expressions. It is always possible to check whether or not an expression is an atom or a composite, by acting on it with the built-in predicate AtomQ. For instance:

```
ClearAll["Global`*"];
{AtomQ[x], AtomQ[Sin[x]], AtomQ[1 + I * 2], AtomQ[2 / 3]}
```

```
{True, False, True, True}
```

■ 1.1.2 *Mathematica* normal (composite) expressions

Every normal expression (composite) is built according to a universal pattern:

```
expr[el1, ..., eln]
```

Here it is required that some symbol < expr > is present (it can itself be a normal expression, not necessarily an atom), as well as the single square brackets. Inside the square brackets, there can be zero, one or several comma-separated elements <el1>,...,<eln>. These elements themselves can be either atoms or normal expressions. In an expression Sin[x], <expr> is Sin, and there is a single element <x>, which is atom (as long as x is not defined as something else, but this already has to do with expression evaluation and will be discussed below). It is clear that an arbitrary *Mathematica* expression must have a tree-like structure, with the branches being normal (sub)expressions and leaves being atoms.

■ 1.1.3 Literal equivalents of built-in functions, and FullForm command

As a consequence, any built-in command/function in *Mathematica* has a literal/string equivalent (so that it can be represented in the above uniform way). This is most easily seen with the help of the built-in function FullForm, which shows the internal representation of any object/expression, in the way it is really "seen" by the kernel. For instance:

```
{z * Sin[x + y], FullForm[z * Sin[x + y]]}
```

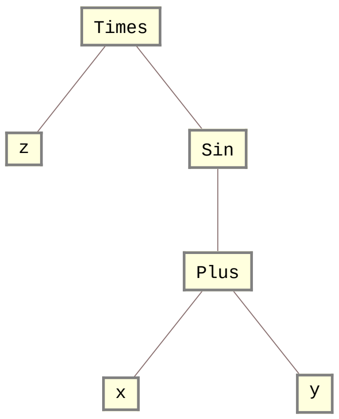
```
{z Sin[x + y], Times[z, Sin[Plus[x, y]]]}
```

The second expression in the curly braces is equivalent to the first one, but explicitly shows the structure described above.

1.1.4. All normal expressions are trees - TreeForm command

That it is a tree, can be seen most clearly with another built-in command TreeForm:

```
TreeForm[z * Sin[x + y]]
```



Since it is a tree, it is possible to index and access the subexpressions. In the following example <expr> is Times (the multiplication command):

```
a = z * Sin[x + y];
FullForm[a]
```

```
Times[z, Sin[Plus[x, y]]]
```

1.1.5. Heads of expressions and the Head command

In general, an expression outside the square brackets has a name - it is called a head of expression, or just head. There is a built-in function with the same name, which allows to obtain the head of an arbitrary expression. For example:

```
Head[a]
```

```
Times
```

A head of an expression may be either an atom or a normal expression itself. For example :

```
Clear[b, f, g, h, x];
b = f[g][h][x];
```

```
Head[b]
```

```
f[g][h]
```

```
Head[f[g][h]]
```

```
f[g]
```

```
Head[f[g]]
f
```

Every expression has a head, even atoms. Heads for them are String, Symbol, Integer, Real, Rational and Complex. For instance :

```
{Head[f], Head[2], Head[Pi],
 Head[3.14], Head["abc"], Head[2 / 3], Head[1 + I]}
{Symbol, Integer, Symbol, Real, String, Rational, Complex}
```

1.1.6 Accessing individual parts of expressions through indexing

One can access also the internal parts of an expression (those inside the square brackets), by using indexing (Part command). The following example illustrates this.

```
{a[[0]], a[[1]], a[[2]], a[[2, 0]], a[[2, 1]], a[[2, 1, 0]],
 a[[2, 1, 1]], a[[2, 1, 2]]}

{Times, z, Sin[x + y], Sin, x + y, Plus, x, y}
```

We have just deconstructed our expression to pieces. In fact, we started from the "stem" and then went down along the "branches" to the "leaves" of the tree which we have seen above with the TreeForm. We see that the addresses (index sequences) which end with zero give the Heads of the subexpressions - this is a convention. In principle, any complex expression can be deconstructed in this way, and moreover, one can change its subexpressions.

1.1.7 Levels of expressions and the Level command

It is also possible to get access to the branches (subexpressions) which are at the certain distance (level) from the "stem". This is achieved by using a built-in Level command. Consider an example:

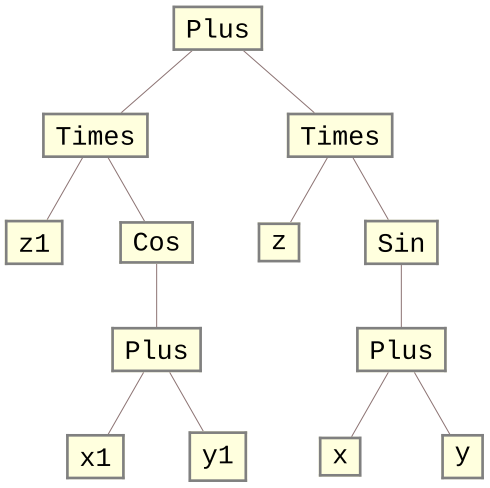
```
Clear[a];
a = z * Sin[x + y] + z1 * Cos[x1 + y1]
z1 Cos[x1 + y1] + z Sin[x + y]
```

Here is its full form :

```
FullForm[a]
Plus[Times[z1, Cos[Plus[x1, y1]]], Times[z, Sin[Plus[x, y]]]]
```

Here is its tree form :

```
TreeForm[a]
```



And these are the levels of the tree :

```
Level[a, {0}]
Level[a, {1}]
Level[a, {2}]
Level[a, {3}]
Level[a, {4}]

{z1 Cos[x1 + y1] + z Sin[x + y]}

{z1 Cos[x1 + y1], z Sin[x + y]}

{z1, Cos[x1 + y1], z, Sin[x + y]}

{x1 + y1, x + y}

{x1, y1, x, y}
```

Level[a, {n}] gives all branches (or leaves) which have a distance of n levels down from the "stem". If however we need all branches that have n levels of sub - branches (or leaves), then we use a negative level Level[a, {-n}] :

```
Level[a, {-1}]
Level[a, {-2}]
Level[a, {-3}]
Level[a, {-4}]

{z1, x1, y1, z, x, y}

{x1 + y1, x + y}

{Cos[x1 + y1], Sin[x + y]}

{z1 Cos[x1 + y1], z Sin[x + y]}
```

Notice that negative levels generally can not be reduced to positive levels - they are giving in general different types of information. What we have just described is called the Standard Level Specification in *Mathematica*. Many more built - in commands accept level specification as one of the arguments (often an optional one).

Any function can be used also in its literal equivalent form. For instance :

```
{Plus[1, 2, 3, 4], Times[1, 2, 3, 4]}
{10, 24}
```

1.2 Second principle: pattern-matching and rule substitution

Another fundamental principle is so - called pattern - matching, which is a system to match rules and expressions - without it *Mathematica* would not know when to apply which rule. It is based on syntactic rather than semantic comparison of expressions. The main notions here are those of rules and patterns.

1.2.1 Rewrite Rules

```
Clear[a, b, c, d, e];
```

A typical *rule* looks like

```
a → b
```

where in general < a > and < b > are some *Mathematica* expressions. The rule just says: whenever <a> is encountered, replace it by <b>. For example:

```
{a, c, d, c} /. a → b
{b, c, d, c}
```

(the < /. > symbol is a rule replacement command, to be covered later).

A *pattern* is essentially any expression with some part of it replaced by "blank" (Blank[]), which is a placeholder for any expression - that is, instead of that part there can be anything (this is somewhat oversimplified). The literal equivalent for Blank[] is the single underscore ("_") symbol. For instance, f[x_] means f[anything].

1.2.2 An example of a simple pattern-defined function

```
Clear[f];
f[x_] := x^2;
{f[2], f["word"], f[Newton]}
```

$\{4, \text{word}^2, \text{Newton}^2\}$

In this example, the result is as shown because the definition of the function < f > is really just a substitution rule f[anything] -> (anything)^2.

■ 1.2.3 Functions are really rules : DownValues command.

To see the internal form of this rule - how it is stored in the rule base - one can use the built-in DownValues command. With its help we see:

```
DownValues[f]
```

```
{HoldPattern[f[x_]] :→ x^2}
```

We will talk later about the meaning of the HoldPattern function. The pattern x_ is the most simple pattern. There can be more complex patterns, both syntactically and also because patterns may have conditions attached to them, which ensure that the pattern will match only if the condition is satisfied (conditional patterns). We will cover them in detail later.

■ 1.2.4 Example of a function based on a restricted pattern

Now let us give an example: we will restrict our function <f> to operate only on integers.

```
Clear[f];
f[x_Integer] := x^2;
{f[2], f[Pi], f["word"], f[Newton]}
```

```
{4, f[π], f[word], f[Newton]}
```

In this case, we introduced a more complex pattern x_Integer.

■ 1.2.5 A bit about evaluation

On this example we see that if there is no rule whose pattern (left hand side of the rule) matches a given expression, *Mathematica* returns the expression unchanged. This is at the heart of its evaluation method: to any entered expression, all rules which are in the global rule base at the moment of evaluation, are applied iteratively. Whenever some rule applies, an expression is rewritten and the process starts over.
At some point, the expression becomes such that no rule can be applied to it, and this expression is the result. Since the rule base contains both system and user-defined rules (with the latter having higher priority), it gives great flexibility in manipulation of expressions.

■ 1.2.6 Patterns allow for multiple definitions of the same function

As another starting example, let us define a function which is linear on even numbers, quadratic on odd numbers and is a Sin function for all other inputs:

```
Clear[f];
f[x_?EvenQ] := x;
f[x_?OddQ] := x^2;
f[x_] := Sin[x];
```

Here is an example of its execution on various inputs :

```
{f[1], f[2], f[3], f[4], f[3 / 2], f[Newton], f[Pi]}
```

```
{1, 2, 9, 4, Sin[3/2], Sin[Newton], 0}
```

For the record, built - in functions OddQ and EvenQ are predicates which return True if the number is odd (even) and False otherwise :

```
{EvenQ[2], EvenQ[3], OddQ[2], OddQ[3]}
```

```
{True, False, False, True}
```

If nothing is known about the object, they give False :

```
{EvenQ[Newton], OddQ[Newton]}
{False, False}
```

▪ 1.2.7 Non - commutativity of rules substitution

Let us look at the last of the 3 definitions of < f > in the above example. It implies that any input object has to be replaced by its Sine. Naively, this would mean that we should have obtained Sine-s of all our expressions, but this did not happen. The point is that the sequential rule application is non-commutative: first of all, the way rules are applied is such that once the first rule that applies is found, only this rule is applied, and other possibly matching rules are not tried on a given (sub)expression, in a single "run" of the rule application. Second, if several rules match an expression, the first applied rule rewrites it so that (some) of other rules don’t match it any more. Therefore, the result depends on the order in which the rules are applied. *Mathematica* applies rules to expressions sequentially. Since the rule with the Sine function was defined last, it should mean that it has a chance to apply only to inputs whose form did not match patterns in the first two rules.

▪ 1.2.8 Automatic rule reordering

What is less trivial is that in this example we would get similar behavior even if we defined this rule first:

```
Clear[f];
f[x_] := Sin[x];
f[x_ ? EvenQ] := x;
f[x_ ? OddQ] := x^2;
{f[1], f[2], f[3], f[4], f[3 / 2], f[Newton]}
{1, 2, 9, 4, Sin[3/2], Sin[Newton]}
```

To see the order in which the rules are kept, we again use DownValues :

```
DownValues[f]
{HoldPattern[f[x_ ? EvenQ]] :> x,
 HoldPattern[f[x_ ? OddQ]] :> x^2, HoldPattern[f[x_]] :> Sin[x]}
```

We see that the rule with Sine again is at the end, despite having been defined first. The reason is that *Mathematica* pattern-matching engine has a built-in rule analyzer which sorts the rules such that more general rules come after more specific ones, *when it can determine it*. This is not always possible to do automatically (and not always possible to unambiguously do at all), so in general the programmer should take care of it. But in practice, it is seldom needed to manipulate the rules by hand.

1.3 Third principle: expression evaluation

The last example brings us to the third principle: the principle of expression evaluation and the rewrite rules (global rule base). It tells the following: when *Mathematica* encounters an arbitrary expression, it checks its global base of rewrite rules for rule(s) which correspond to a given expression (or, it is said, match the expression). A typical rewrite rule looks like object1 -> object2. If such a rule is found, for expression or any of the subexpressions (actually, normally in reverse order), the (sub) expression is rewritten, and the process starts over. This process goes on until no further rule in the global rule base is found which matches the expression or any of its parts. When the expression stops changing, it is returned as the answer. Please bear in mind that the picture just described is a great oversimplification, and the real evaluation process is much more subtle, although the main idea is this.

The global rule base contains both rules built in the kernel and rules defined by the user. User-defined rules usually take precedence over the system rules, which makes it possible to redefine the behavior of almost any built-in function if necessary. In fact, all assignments to all variables and all function definitions are stored as some type of global rules, in the rule base. In this sense, there is no fundamental difference between functions and variables (although there are technical differences).

As a result of this behavior, we get for instance such result:

```
FullForm[Sin[Pi + Pi]]


0
```

The reason is that inside the kernel there are rules like Plus[x,x]->2 x, Sin[2*Pi]->0, and because the evaluation process by default starts with the innermost sub-expressions (leaves), i.e., from inside out, it produces 0 before the FullForm has any chance to "look" at the expression. The internal evaluation dynamics can be monitored with the Trace command:

```
Trace[FullForm[Sin[Pi + Pi]]]
{{{π + π, 2 π}, Sin[2 π], 0}, 0}
```

Summary

To summarize, we have described briefly the main principles of *Mathematica* and along the way gave examples of use of the following built-in functions: AtomQ, Head, FullForm, TreeForm, Level, Plus, Times, Trace, DownValues, OddQ, EvenQ.

II. Elementary operations

■ 2.1 Introduction

In this chapter we will discuss some basic operations with variables such as assignments, and also some procedural control flow structures, such as conditionals and loops.

■ 2.2 Symbols and variables

In *Mathematica* , variables are (possibly composite) symbols that can store some definitions. More precisely, as variables can be used expressions which can be the l.h.s. of global rules not containing patterns (consider this as our definition of variables. What it means is explained below).

■ 2.2.1 Legitimate symbol names

An immediate comment on legal symbol names is due here. Any name which has a head Symbol and is not a name used by the system, is legal to associate some rules (global definitions) with - in other words, a legal name for a variable or a function. It can not start with a number, but can contain numbers. It can contain both capital and small letters, and *Mathematica* is case-sensitive, so names like <totalsum> and <totalSum> are considered different. There are several special symbols that should not be used in symbol names, such as @,#,$,%,^,&,*,!,~,‘. What may be surprising for C programmers is that the underscore <_> should not be used either - it has a different purpose in *Mathematica.*

If you are in doubt whether your symbol is legitimate, you can always use the **Head[-Unevaluated[yoursymbol]]** command to check its head: if it is <Symbol>, you are in business. The reasons why you need <Unevaluated> are a bit involved to discuss now, but basically it guarantees that the Head function tests the head of your original symbol rather than what it may evaluate to. For instance, you decide to use <a&&True> symbol. This is the result of using Head:

```
Head[a && True]
Symbol
```

We happily try to assign something to it :

```
a && True = b
Set::write : Tag And in a && True is Protected. ≫
b
```

and get an error message.

What happens is that our symbol is really a logical And of <a> and True, and it evaluates to <a> when we try to use Head function - so the head seems to be Symbol. But in the assignment, the original symbol is used, and its head is And. For reasons which will become clear soon, this leads to an error. Using Head[-Unevaluated[symbol]] reveals that our symbol is bad:

```
Head[Unevaluated[a && True]]
And
```

This behavior will become clear after we cover more material.

```
a && True = 1
```

Set::write : Tag And in a && True is Protected. ≫

```
1
```

It is also a good habit to start the name of any symbol you use (variable or function) with a small letter, since then there is no chance of colliding with a built-in symbol - all built-in symbols in *Mathematica* start with a capital letter or a special symbol such as $.

■ 2.2.2 Getting information about symbols

For every symbol with the head Symbol (as tested by Head[Unevaluated[symb]] command, see above), it is possible to display the information that the system has on that symbol currently. This includes possible rules (definitions) associated with the symbol, some other properties possibly attached to the symbol, such as Attributes, and for system symbols brief explanations of their purpose and use.

To display the global rules associated with the symbol, one uses either the question sign in front of the symbol for a brief info, or either a double question sign or the Information command, for more details. For example, here we define a global variable <a>:

```
a = 3;
```

This is how we inspect < a > :

```
? a
```

```
Global‘a
```

Or, which is the same in this case,

```
?? a
```

```
Global‘a
```

If we inspect a symbol which has not been introduced to the system yet (it is said not to be in the symbol table), we get a message :

```
? c
```

Information::notfound : Symbol c not found. ≫

If we inspect some built - in symbols, we usually get a quick summary of their functionality :

```
? Map
```

Map[*f*, *expr*] or *f* /@ *expr* applies *f* to each element on the first level in *expr*.
Map[*f*, *expr*, *levelspec*] applies *f* to parts of *expr* specified by *levelspec*. ≫

We get more here by using ?? (or, Information[]):

```
?? Map
```

Map[*f*, *expr*] or *f* /@ *expr* applies *f* to each element on the first level in *expr*.
Map[*f*, *expr*, *levelspec*] applies *f* to parts of *expr* specified by *levelspec*. ≫

```
Attributes[Map] = {Protected}

Options[Map] = {Heads → False}
```

2.2.3 "Proper" variables and OwnValues

By "proper" variables we will mean variables with names being symbols (in the sense described above, with the head Symbol), which are used to store values. Due to symbolic nature of *Mathematica*, these values can be of any type, either atoms or normal expressions (there is no notion of "type" in *Mathematica* as such - see below).

The built - in function which reflects the possible assignment made to a "proper" variable is called **Own-Values**. For example :

```
a = 3;
OwnValues[a]
{HoldPattern[a] :→ 3}
```

The equal sign here represents an assignment operator, which we will cover shortly.
Another way to characterize the "proper" variables is that their definitions are stored in OwnValues. One particular property of OwnValues is that for a given symbol there can be either no global rule or just one global rule (but not more) stored in OwnValues, and OwnValues only store rules associated with real symbols (with the head Symbol). This is another reason why I call this type of variables "proper".

2.2.4 Indexed variables and DownValues

In addition to these, there are other objects which can also store values and sometimes be interpreted as variables. In particular, consider the following assignments :

```
b[1] = 1;
b[2] = 4;
b[3]  = 9;
```

This looks like array indexing, but it is not (see section 2.9.4). We can check whether or not these definitions are stored as OwnValues :

```
OwnValues[b]
{}
```

They are not. Instead, they happen to be stored as **DownValues** (another type of global rules) :

```
DownValues[b]
{HoldPattern[b[1]] :→ 1, HoldPattern[b[2]] :→ 4, HoldPattern[b[3]] :→ 9}
```

We see that there can be many global rules stored in DownValues and associated with the same symbol (<b> here). In general, DownValues are used to store the function definitions. Thus, one way to interpret the above assignment is that we defined a function < b > on 3 specific values of the argument. Sometimes, however, it is more convenient to interpret b[i] as a set of "indexed variables" - particularly when there is no general pattern - based definition for < b > (if this is somewhat unclear now, wait until chapter IV on functions). Thus, in some cases we will interpret these composite objects (notice, their head is not <Symbol> - in this case it is <b>) as "indexed variables".

Indexed variables are quite useful in many circumstances. The indices for them can be any *Mathematica* expressions (not necessarily numbers, like in this example), and therefore one possible use of them is to implement data structures such as Python dictionaries. Internally, hash tables are used to make indexed variables efficient, and thus, while there is no direct support for hash-tables in *Mathematica* (in version 6 there is a Hash built-in function though), one can actually use the internal hash tables just as well through indexed variables.

2.2.5 Composite variables and SubValues

Finally, there can be cases like the following :

```
Clear[a, b, c, x, y, z];
a[b][1] = x;
a[b][2] = y;
a[c][1] = z;
```

Such definition is legal (the Clear command is used to clear variables and will be covered in a moment). However, you can check that this definition is stored neither in OwnValues nor in DownValues of < a > or < b > . Note also that the head of such a "variable" is a composite object itself:

```
Head[Unevaluated[a[b][1]]]
```

```
a[b]
```

The definitions like above are stored as yet another type of a global rule, called a SubValue (**SubValues** command) :

```
SubValues[a]
```

```
{HoldPattern[a[b][1]] :→ x,
 HoldPattern[a[b][2]] :→ y, HoldPattern[a[c][1]] :→ z}
```

We see that there can be more than one global rule stored in SubValues, similarly to DownValues. Note that SubValues associate these definitions with <a>, and in general with the outermost head of the composite head of an expression (this is called "symbolic head"). While this can also be considered as a variable in the sense that it can store values, it is rather uncommon (especially when used as a variable) and I would discourage the use of such "variables" except very special circumstances (such as, for example, to implement nested hash - tables).

2.2.6 Clearing variables

Since the rules for variables and functions are stored in the global rule base, it is essential for a "clean" work to make sure that we work with "clean" variables. This is achieved by using the **Clear[var]** command. In particular, this command removes all the definitions associated with a variable and stored as OwnValues, DownValues, SubValues or other types of global rules (there are only 3 more - UpValues, NValues and FormatValues, but these types of rules are more advanced and we will not talk about them at this time). It is a good habit to clear any symbols you use right before they are defined. Many examples of using Clear will follow.   For now, just a simple example:

These are the current definitions associated with a symbol < b > :

```
?b
```

```
Global‘b

b[1] = 1

b[2] = 4

b[3] = 9
```

We now Clear < b > :

```
Clear[b];
?b
```

```
Global‘b
```

We see that the definitions were cleared, however the symbol remained in the symbol table.

One moment to mention here is that only symbols (with the head Symbol) or strings can be arguments of Clear (if you are interested when and why strings can be used, consult the *Mathematica* Help or *Mathematica* Book). In particular, an attempt to clear our composite variable in this way will fail :

```
Clear[a[b]]
```

```
Clear::ssym : a[b] is not a symbol or a string. ≫
```

One has to be more selective in this case, because many different variables are associated with the symbol < a > . In such cases, built - in **Unset** is used, or its short - hand notation equal - dot < =. >

```
a[b][1] =.

?a
```

```
Global‘a

a[b][2] = y

a[c][1] = z
```

We see that we removed the definition for the a[b][1] variable.

The Clear command removes the rules associated with a symbol, but not some other symbol properties, such as Options or Attributes. Consider the following code :

```
Options[b] = {Heads → True};
Attributes[b] = HoldAll;
```

We have assigned some properties to a symbol < b > . Now we Clear it :

```
Clear[b];
?b
```

```
Global‘b

Attributes[b] = {HoldAll}

Options[b] = {Heads → True}
```

To clear all properties associated with the symbol, use **ClearAll** :

```
ClearAll[b];
?b
```

```
Global‘b
```

ClearAll, however, does not remove the symbol from the symbol table either. Sometimes it is desirable to remove the symbol completely, so that it will also be removed from the symbol table. This is done by the **Remove** command:

```
Remove[b];
?b

Information::notfound : Symbol b not found. ≫
```

▪ 2.2.7 Does a variable have a value? ValueQ

In *Mathematica* , it is quite legal for a variable (symbol or expression) not to have any value, or, put differently, not to be a l.h.s. of any global rule present currently in a system. For example :

```
b
b
```

Since < b > here has no value yet, it is returned back (it is said to evaluate to itself).

For any given variable, or expression in general, one may ask whether or not it has a value, or in other words, whether or not it is a l.h.s. of some global rule present in the system. Of course, on way would be to check all the ... Values, but there exists a better way : use a built - in ValueQ predicate. For instance :

```
{ValueQ[a[b][1]], ValueQ[a[b]]}
{True, False}
```

Notice that while a[b][1] has a value, this does not mean that a[b] necessarily has one, as ValueQ indicates. In this particular case, a[b] has to be treated as a composite name of an indexed variable.

▪ 2.2.8 Assignments attach rules to Symbol-s

From this discussion it may seem that there are a lot of different cases to learn about variable definitions. But at the end of the day, in all cases described above, assignments attach global rules to symbols with the head Symbol. For OwnValues, this is just the variable itself, for DownValues this is a head of an indexed variable, and for SubValues this is a symbolic head of the composite head of the variable. So, for example, to clear any particular definition, one has to locate the symbol to which this definition is attached, and either Clear this symbol or use Unset if one needs to be selective, rather than trying to call Clear on composite expressions like in our failed attempt above.

2.2.9 Summary

The variables in *Mathematica* can be either symbols or normal expressions and can store any *Mathematica* expressions. The names of symbols can contain letters and numbers, but can not start with a number or contain certain special characters, underscore included. One can use Head[Unevaluated[symb]] command to check whether or not the symbol is legitimate (has a head Symbol). One should not use variables with names coinciding with some of the many system symbols. Starting a symbol name with a small letter will ensure no conflicts with the system symbols.

There are several commands in *Mathematica* which display the information about the variables, such as < ?> and <Information>.

All variable definitions are global rules. Normally, variable names are just symbols (head Symbol). The definitions for such variables are rules called OwnValues and stored in OwnValues[symbol]. The second (rather rarely used) type of variables are indexed variables. They have head being another simple symbol (like a[1] has a head < a >), rather than Symbol, and are stored in DownValues attached to their head. In some rare cases one can also introduce variables with composite heads. These are stored in SubValues attached to their symbolic head. The latter two types of variables are usually used in more special circumstances, since DownValues and SubValues are normally associated with functions rather than variable definitions.

You will be better off not using DownValues or SubValues - based variables until you understand exactly when and for which purpose they are beneficial.

Once the global definition is given to a variable, it remains there until another definition of the same type is entered to replace it, or until it is cleared.

To clear the "normal" (OwnValue) variable definition, the Clear command is used. To clear also all other properties possibly associated with the variable, use ClearAll. If you also need the symbol (name of the variable) to be removed from the symbol table, use Remove. All these commands can not be used on composite variables (DownValue-based or SubValue-based). To clear such variables, use Unset.

■ 2.3 Dynamic data typing

Mathematica is a dynamically typed language, which means that the type of the variable is inferred when it is defined, and a single variable may be used as a placeholder for different types of data during the same *Mathematica* session (although this is not a best thing to do). In fact, there is no notion of type similar to other languages - it is replaced by notion of atoms and normal expressions as explained in the previous chapter.

Any newly entered object is considered as a symbolic entity. Whenever it is entered, the symbol table is searched for it. If it is there, a new reference to it is created. If not, a new entry in the symbol table is created. If the object is new, or such that no rules immediately apply to it, it is returned:

```
Clear[a];
a
```

```
a
```

```
Sin[a]
Sin[a]
```

In the last example, Sin is a built-in function, however no rules are associated with Sin[a] or <a>, so the expression returns.

2.4 Assignments

2.4.1 Immediate and delayed assignments: Set and SetDelayed

There are two assignment commands in *Mathematica*: immediate and delayed assignment.

The immediate assignment is performed with the equal sign (=), say x = y, which means "assign to x the value that y has right now". This command has a literal equivalent Set: we could equivalently write Set[x,y]. For delayed assignment, one has to use (:=) (semicolon plus an equal sign), say x:=y. The literal equivalent is the SetDelayed command, for instance SetDelayed[x,y]. This means "add to the global rule base a rule, which will substitute x every time that x is encountered, by the value that y will have at that moment". So, with this kind of assignment, the right hand side is not evaluated at the moment of assignment, but is re-evaluated every time when the left-hand side appears somewhere in the program. This is the main difference between the two assignment operators, since with Set, the l.h.s. is assigned the value that the r.h.s. has at the moment of assignment, "once and for all".

2.4.2 The difference between Set and SetDelayed : an example

Here is an example:

```
Clear[a, b];
a = Random[Integer, {1, 10}];
Do[Print[a], {5}]
```

```
2
2
2
2
2
```

```
b := Random[Integer, {1, 10}];
Do[Print[b], {5}]
```

```
2
6
6
10
2
```

In both cases, an object (a or b) is assigned a random integer in the range 1 -10. But in the former case, it is computed at the moment of assignment and then is attached to <a>, while in the latter case it is recomputed afresh every time that b is called. We have also used one of the looping constructs Do, which we will cover shortly.

■ 2.4.3 Chain assignments

Chain assignments are possible for both Set and SetDelayed, but while the meaning of the former is the same as in C, the meaning of the latter is really different. Let me illustrate this point.

Here we assign the variables a,b and c the value 2:

```
Clear[a, b, c];
a = b = c = 2;
{a, b, c}
```

```
{2, 2, 2}
```

Let us try the same with SetDelayed:

```
Clear[a, b, c];
a := b := c := 2;
{a, b, c}
```

```
{Null, Null, 2}
```

The Null output means "no output". This is because while the Set operator is actually an expression in the C sense and returns the value - that of the r.h.s, SetDelayed is an operator but a statement rather than an expression in the C sense, so it does not return a value. But what is important here is that neither <a> nor <b> received a "value" in the normal sense as variables (formally they did, according to our discussion of variable above, but what they received I would not call a "value"). Rather, they are triggers which trigger certain assignments. When <a> is called, this means "associate with b the rule that whenever b is called, the value of the r.h.s. of this rule as it is at this moment is returned". But the r.h.s of the rule for <b> is an assignment (of 2) to <c>, which means that after <a> is called just once, the values of 2 will be assigned to <c> every time that b is called. We can watch this once again to check the above statements:

```
Clear[a, b, c];
a := b := c := 2;
```

```
?a
```

```
Global‘a

a := b := c := 2
```

```
?b
```

```
Global‘b
```

```
?c
```

```
Global‘c
```

We now call a:

```
a
```

The new definitions:

```
?b
```

```
Global‘b

b := c := 2
```

```
?c
```

```
Global‘c
```

Now we call b:

```
b
```

The new definitions:

```
?b
```

```
Global‘b

b := c := 2
```

```
?c
```

```
Global‘c

c := 2
```

So, it is not clear where this construction may effectively be used, but it is clearly very different from what one may naively expect.

■ 2.4.4 Don’ t use SetDelayed with more than two arguments

Notice by the way, that although SetDelayed in a literal form can also accept more than 2 arguments, the result will not be what one would probably expect. Namely, SetDelayed[a,b,c] is not the same as a:=b:=c. Rather, it attaches to a the rule that a be replaced by a sequence of b and c:

```
Clear[a, b, c];
SetDelayed[a, b, c];
?a
```

```
Global‘a

a := Sequence[b, c]
```

The head Sequence is a rather special head in *Mathematica*, and is used to represent "no head". It is needed since according to one of the main principles, everything is an expression, and as such must have some head or be an atom (atoms also have heads but those are fixed by convention rather than deter-mined from syntax). However, the head Sequence is special since normally it "disappears" inside other heads:

```
Clear[f, a, b, c];
f[Sequence[a, b, c]]
```

```
f[a, b, c]
```

The reason it did not disappear inside SetDelayed is that SetDelayed is one of the exceptions (it has to do with more general issue of function attributes and will be covered later).

So, the final conclusion: don’t use the chain assignment with SetDelayed, unless this is indeed what you want to do - chances are that the result will be very different from what you expect.

2.4.5 Set and SetDelayed : when which one is used

Normally one uses Set (=) to define variables, and SetDelayed (:=) - to define functions, for which the recomputation of the r.h.s. on a changed argument is a natural operation. However, in this respect *Mathematica* differs from other programming environments in that the distinction between functions and variables is achieved not on the level of keywords such as <function>, or specific forms of functions declaration, but essentially by the assignment operator that have been used to define the symbol (this is somewhat oversimplified), and also to some extent by a type of global rule associated with the symbol (but again, it is decided based on the syntactic form of the symbol, but at the moment of assignment). This allows to work with functions and variables on equal grounds. This lack of fundamental distinction between functions and variables is at the heart of the functional style of programming, which is one of the most efficient programming styles in *Mathematica* and which we will use a lot in the later chapters.

■ 2.5 Equality checks

There are two operators in *Mathematica* which can be used to perform equality checks. They perform checks on semantic equality (equality by value) and syntactic equality (equality by syntactic form of expression).

■ 2.5.1 Equal

The first operator has a short-hand notation as double equal sign (like in C): <==>, for instance x == y, and a literal equivalent Equal[x,y]. It is associative so that it is legitimate to write say x==y==z. This is different from C, where the above will not work as desired since y==z would evaluate to 1 or 0, and therefore x will be then compared to the result of comparison of y and z rather than to y or to z (which will probably be a bug). The literal equivalent of the last example would be Equal[x,y,z]. We can see that Equal can take many arguments, which is often handy. For example:

```
Clear[a, b, c];
a = b = c = 2;
d = 3;
{a == b, b == d, a == b == c, Equal[a, b, c], Equal[a, b, c, d]}
```

```
{True, False, True, True, False}
```

■ 2.5.2 Beware: Equal may return "unevaluated"

Equal works on any *Mathematica* expression, atom or a normal expression. However, in general, for arbitrary l.h.s. and r.h.s, Equal evaluates to itself because *Mathematica* can not determine whether or not the expressions have the same value. For example:

```
Sin[x]^2 + Cos[x]^2 == 1
```

```
Cos[x]^2 + Sin[x]^2 == 1
```

This does not mean that *Mathematica* can not simplify the l.h.s. :

```
Simplify[Sin[x]^2 + Cos[x]^2]
```

```
1
```

It just means that by default it will not do so, unless instructed explicitly. In such cases, Equal returns unevaluated (not in the sense that it has not been evaluated at all, but in the sense that evaluation resulted in the original expression). This makes a lot of sense in a symbolic environment such as *Mathematica*, since at some point, new rules may be added, then expression will change and perhaps then Equal will evaluate. For instance:

```
Simplify[Sin[x]^2 + Cos[x]^2 == 1]
True
```

■ 2.5.3 Equal is used by built-in solvers to build equations

The above behavior of Equal - the fact that it evaluates to itself whenever equality can not be determined (on different symbolic expressions on different sides), is exploited in various built - in functions which receive equations as their arguments, such as Solve, DSolve, etc. The equation is written as for instance x^2 - 3 x + 2 == 0:

```
Solve[x^2 - 3 x + 2 == 0, x]
{{x → 1}, {x → 2}}
```

■ 2.5.4 Assigning the result of Equal to a variable, and more on evaluation

Since Equal[a, b], as everything else, is a *Mathematica* expression, we can assign some variable the value of this expression. As an example, consider the following statement :

```
Clear[x, test];
test = Sin[x] == 0
Sin[x] == 0
```

Let us now give < x > a value :

```
x = Pi;
test
True
```

We see that initially test was unevaluated, but once x received a global value, test evaluated to True. What may look surprising however, is that internally < test > still has the previous definition, rather than < True > :

```
?test
Global‘test

test = Sin[x] == 0
```

In particular, if we now Clear our < x > variable, the result will again be a symbolic expression :

```
Clear[x];
test
Sin[x] == 0
```

This means that the variable < test > in fact behaves like a function of x (although this is not a proper way to define functions). But more importantly, this explains to us part of the evaluation mechanics. First of all, the definition of <test> represents a rule in the global rule base. This can be seen by using the OwnValues function, which describes the definitions of variables (atoms):

```
OwnValues[test]
{HoldPattern[test] :→ Sin[x] == 0}
```

Secondly, the way any global rule obtained by using the Set command is added, is that the r.h.s. is evaluated at the moment of definition, and the result added to the rule base. If some variables (x in this case) did not have a value at the moment of assignment, they will be added in their symbolic form. But once the rule has been added to the global rule base, it will not change in any way (unless it is removed or manipulated by hand later), regardless of possible changes in the variables used in the r.h.s., which happened after the rule has been added. That's why in our example the definition of <test> remained unchanged despite changes in the value of <x>.

It is very important to realize that rules in the global rule base exist completely independently of the global state of the system, in the sense that they can be only added or removed from the rule base, but do not change regardless of changes of the global state. If we think about it, this is the only way it can be, since the global state is itself determined by the state of the rule base. So, what we have just discussed may be rephrased like this: different rules added to the global rule base do not interact with each other during their "stay" in the rule base, but only during the evaluation of some expression. Had this not been so, and we would have no way to predict the outcome of the evaluation, since some rules would change other rules and the result would be different depending on the order in which they are applied.

■ 2.5.5 The role of side effects

The symbolic nature of *Mathematica* may make one think that if any symbolic expression entered is the same on the l.h.s. and r.h.s. of the comparison operator, one would always get True. But this is not always so, in particular when the expression has side effects such as assignments. As an example, consider a function which increments its argument (ignore the code, we will cover it later).

```
Clear[a, inc];
a = 5;
inc[x_] := x = x + 1;
Attributes[inc] = {HoldAll};
```

Now, check this :

```
inc[a] == inc[a]
False
```

■ 2.5.6 A digression: standard and non-standard evaluation

What happened is very easy to see by using Trace command :

```
a = 5;
Trace[inc[a] == inc[a]]
{{inc[a], a = a + 1, {{a, 5}, 5 + 1, 6}, a = 6, 6},
 {inc[a], a = a + 1, {{a, 6}, 6 + 1, 7}, a = 7, 7}, 6 == 7, False}
```

We see that first, inc[a] on the left evaluated, which resulted in the value 6 but also incremented a as a side effect. Next, the r.h.s. evaluated, resulting in a value 7 and once again incrementing a. Thus, when Equal actually tested the expression, it looked like 6==7 which of course resulted in False (in fact, this example is not unique to Mathematica. Similar situation would also occur in C in the same setting (change inc to ++), and there we would not even know which side of the comparison will evaluate first - this is up to a compiler).

I used this example to illustrate several points. First, we saw that subexpressions inc[a] evaluated before the expression containing them (Equal in this case) evaluated. This is a general rule of *Mathematica* evaluation - it normally starts from leaves and goes from inside out to branches and then to root. This is called standard evaluation. However, you may have noticed the inconsistency: by the same logic, <a> should have been evaluated first, and thus the function < inc > should have had no chance of changing the value of < a > whatsoever. This is because it had to always get a number (current value of <a>), since <a> is a leaf for < inc >, and thus had to be evaluated before < inc[a] > . And you will be absolutely correct - normally this does not happen, and this behavior (when rules associated with expressions are applied before rules associated with sub-expressions) is called non-standard evaluation and in this case was induced by the second line in the definition of <inc> (HoldAll attribute - to be discussed later).

Both standard and non-standard evaluation are equally important for the functioning of *Mathematica*. Most built-in functions evaluate their arguments in the standard way, but many crucial built-in functions evaluate their argument in the non-standard way. This topic is too advanced for us now, but we will return to it later and consider it in more detail.

■ 2.5.7 Pass by reference semantics - possible to imitate

Another point that this example illustrated is that we may arrange a function to change a value of the input variable such that these changes will remain valid after the function returns. This looks like a pass-by-reference semantics (although this is not entirely true since there are no pointers in *Mathematica*). But in general, such things are rather rarely used in *Mathematica* programming, since usually functions operate on a copy of the variable and the initial variable or expression remains unchanged. Since *Mathematica* functions may return any *Mathematica* expression (including lists containing many results) and memory is allocated dynamically (even variables don’t need to be declared), it is rarely necessary to simulate a pass-by-reference semantics.

■ 2.5.8 SameQ

The above behavior of the Equal operator, and in particular the fact that it may return unevaluated, may sometimes be unsatisfactory. This is so when one needs a definite yes - or - no answer. For this purpose, there exists another comparison operator (but see the caveat below).

The second operator has a short-hand notation as a triple equal sign <===> , for instance x===y, and a literal equivalent SameQ, for instance SameQ[x,y]. Its purpose is to compare expression by their syntactic form rather than value. It gives True when expression are literally the same and False otherwise. In particular, consider the input of our previous example:

```
Clear[x];
Sin[x]^2 + Cos[x]^2 === 1
```

```
False
```

It is interesting that one can also construct examples when Equal gives True while SameQ gives False. For instance, consider this :

```
Pi / 2 == Pi / 2.
True
Pi / 2 === Pi / 2.
False
```

What happens here is that while the value of both sides is the same, syntactically they are different :

```
{Pi / 2, Pi / 2.}
```

$\left\{\frac{\pi}{2}, 1.5708\right\}$

However, the inverse is true : if SameQ gives True, then Equal will also give True.

2.5.9 TrueQ

Caveat: the above example also indicates that sometimes the use of SameQ may be unsatisfactory either, since it may produce False where Equal would eventually produce True. For instance, in the following situation:

```
Clear[a, b, c, d];
a := Equal[c, d];
b := SameQ[c, d];
{a, b}
{c == d, False}
c = Pi / 2; d = Pi / 2.;
{a, b}
{True, False}
```

This means that the purpose of SameQ may in some cases be subtly different from just a replacement for Equal (SameQ, unlike Equal, always evaluates to True or False). For such cases, another operator comes handy: TrueQ. This function gives False whenever its argument does not evaluate to True. So, in the case above (unless we really are interested in aspects in which Pi/2 and Pi/2. are different), the proper thing to do is the following:

```
Clear[a, b, c, d];
a := TrueQ[Equal[c, d]];
```

Check:

```
a
False
c = Pi / 2; d = Pi / 2.;
a
True
```

```
Clear[a, b, c, d];
```

Chain comparisons are also possible, and used every now and then (we will consider a few non-trivial examples of them in later chapters).

As you may have guessed, there are also operators which test whether the sides are unequal rather than equal, or unsame rather than same. You can guess the name of the operators as well - Unequal and UnsameQ, of course. The short - hand notation for Unequal is <!=> , and for UnsameQ - <=!=>. They work in the same way as Equal and SameQ, but of course with opposite results.

■ 2.6 Logical operators

As many other languages, *Mathematica* has several built-in logical operators. We will consider logical AND : short-hand &&, literal equivalent And[], logical OR: short-hand ||, literal equivalent Or[], and the negation NOT (short-hand <!>, literal equivalent Not[]). As you can see, the short-hand notation is the same as in C.

These operators return True or False, but also they may return unchanged if the value of a logical expres-sion can not be determined:

```
And[a, b]
a && b
```

This is perhaps the main difference between these operators in *Mathematica* and other programming languages, of course due to the symbolic nature of *Mathematica*. The situation here is similar to that with If (see below). In case when the definite result is needed, TrueQ can be used:

```
TrueQ[And[a, b]]
False
```

Another difference is that both And and Or can take multiple arguments :

```
And[a, b, c]
a && b && c
Or[a, b, c]
a || b || c
```

Operators And and Or have the same "short-circuit" behavior as their analogs in C : And stops evaluating its arguments and returns False when the first argument which evaluates to False is encountered, and Or stops evaluating its arguments and returns True when the first argument which evaluates to True is encoun-tered. Since normally arguments are evaluated before the function, we immediately conclude that both And and Or use non-standard evaluation.

2.7 Conditionals

2.7.1 The If operator

The If operator has the following syntax :

If[test, oper1, oper2].

Here < test > is a condition being tested. The condition < test > should in principle evaluate to True or False, in order for If to make a choice. If it evaluates to True, the first operator oper1 is evaluated, otherwise a second one. The second operator may be absent, in which case nothing is done for the False outcome (Null is returned).

Since normally the arguments of the function are evaluated before the function itself, while in the case of If the operator which corresponds to True of False branch should only be evaluated after the condition is checked, we conclude that If uses non-standard evaluation.

2.7.2 If may return "unevaluated"

In *Mathematica*, there is a third possibility - that the condition <test> will not evaluate to either True or False. In this case, this is not considered an error, but the whole operator If will return "itself" - a symbolic expression as the answer.

```
Clear[a, b, c];
If[a, b, c]
```

```
If[a, b, c]
```

In case when this is unsatisfactory, one can either use the TrueQ to ensure evaluation, or use an extended version of If with a fourth argument, which will be evaluated when the condition does not evaluate to either True or False :

```
If[a, b, c, d]
```

```
d
```

2.7.3 If returns a value

Another difference with other programming languages is that the whole operator If returns a value. In this sense, it is like a "question" operator in C : a?b : c. Thus, the following is meaningful:

```
Clear[a, b, x, y];
a := If[EvenQ[b], x, y];
```

We check now :

```
a
```

```
y
```

The result is such since by default the predicate EvenQ evaluates to False on an unknown object (see section 1.2.6). Let us change < b > :

```
b = 2;
a
```

```
x
```

And once again :

```
b = 3;
a
```

```
y
```

Notice that < a > gets recomputed every time since it has been defined with SetDelayed (:=). Another example :

```
Clear[a, x, y, test];
a := If[test, x, y];
test := (Sin[x] == Cos[x]);
a
```

```
If[Sin[x] == Cos[x], x, y]
```

Now :

```
x = Pi / 2;
a
```

```
y
```

```
x = Pi / 4;
a
```

$\frac{\pi}{4}$

In these cases, the condition < test > evaluated to False and True respectively, which led to the above values of < a > .

■ 2.7.4 Operators Which and Switch

These operators generalize If to situations where we have to switch between several possibilities. Which is essentially a replacement for nested If statements. Switch in *Mathematica* resembles Switch in C, but differs from it significantly. First, it has a different (extended in a sense) functionality since it works on patterns (patterns are extremely important. We will cover them later). Also, Break[] operator is unnecessary here, and so the fall-through behavior of C Switch is not possible (patterns sort of compensate for this). Finally, in C Switch can be used in more sophisticated ways to put some entry points in the block of code - this is not possible here. Both Which and Switch are well-explained in *Mathematica* Help, and we refer to it for further details regarding them.

2.8 Loops

These constructs are quite analogous to the ones in C, with the exception that the roles of comma and semicolon are interchanged with respect to C. In general, loops are not very efficient in *Mathematica*, and in particular, the most effective in *Mathematica* functional style of programming does not involve loops at all. We will give just a few examples for completeness.

Let us print numbers from 1 to 5 :

2.8.1 For loop

```
For[i = 1, i ≤ 5, i ++, Print[i]];
```

```
1
2
3
4
5
```

If one needs to place several operators in the body of the loop, one can do so by just separating them with a semicolon. The same comment applies to the While loop.

2.8.2 While loop

Or, the same with While :

```
i = 1;
While[i ≤ 5, (Print[i]; i ++)]
```

```
1
2
3
4
5
```

2.8.3 Do loop

And now with Do :

```
Do[Print[i], {i, 5}]
```

```
1
2
3
4
5
```

2.8.4 Side effects induced by loops

In general, most *Mathematica* built - in functions do not introduce side effects, since they operate on a copy of the given expression. In For and While loops however, the index variables (like < i > above) will keep the final value that they had when the loop terminated, as a global value. It is a good practice to localize all loop variables inside one of the scoping constructs available in *Mathematica* (see section 4.8).

On the other hand, the Do loop is a scoping construct by itself, so it localizes the iteration variable. However, the way it does it may be not what one expects (since it effectively uses a Block scoping construct) - it localizes in time rather than in space. I will have more to say about it later (see section 4.8), but "normally" in the following situation:

```
a := i^2;
Do[Print[a], {i, 1, 5}]
```

```
1

4

9

16

25
```

we could expect a symbol < a[i] > printed 5 times (since the definition of < a > uses a global < i > and Do is supposed to localize < i >) . On the other hand, the global < i > does not have any value after < Do > finishes, so <Do> *is* a scoping construct.

```
i
i
```

For clarification of these issues, see section 4.8.

2.8.5 Blocks of operators - the CompoundExpression command

The parentheses in this example actually represent a composite operator, which is legitimate everywhere where the single operator is. Its literal form is **CompoundExpression[oper1; ...; opern]**. The operators have to be separated by semicolons. The value returned by CompoundExpression is the value of the last statement opern. In particular, if we insert a semicolon after < opern > as well, there will be no return value (more precisely, the return value will be Null).

Notice also that the loop Do is not a version of Do - While in C, but just a faster version of For or While, where no condition is checked but one has to know exactly how many iterations are needed.

2.8.6 Local goto statements: Break , Continue, Return

There are statements to realize local Goto within the loop - Break[] and Continue[]. They work in the same way as they work in C. For example, here we will break from the Do loop after 4 iterations :

```
i = 1;
Do[(Print["*"]; If[i ++ > 3, Break[]]), {50}]
```

```
*
*
*
*
```

The Return[] statement can also be used to break from the loops, but it acts differently. We did not cover it yet, but there are three scoping constructs used in *Mathematica* to localize variables - Module, Block and With. If we have a For or While loop inside one of these constructs, then the Break[] statement will break from the loop only, and the code right after the loop (but inside the scoping construct) will start to execute. If we use Return[] however, we will also break from the entire localizing construct which encloses the loop (if there are nested localizing constructs, we break from the innermost one only). Not so for < Do > loop though : it is a localizing construct by itself, so using Return[] we will break out of Do but will remain in whatever localizing construct encloses Do. These comments will become more clear once you get familiar with Module, Block and With (end of chapter IV).

2.8.7 Programs using loops are often inefficient in *Mathematica*

It turns out that programs that use loops are often inefficient in *Mathematica*. This is not so much due to loops themselves being slow in *Mathematica*, as to the fact that certain common programming practices associated with loops are efficient in other languages but not in *Mathematica*.

Let me illustrate this on a simple example. We will compute a sum of the first 100000 natural numbers. We will first do this with a loop, and then show a program written in a functional programming style.

Here is the loop version :

```
Timing[sm = 0;
 Do[sm = sm + i, {i, 100 000}]; sm]
```

```
{0.581, 5 000 050 000}
```

And this is a functional realization :

```
Timing[Apply[Plus, Range[100 000]]]
```

```
{0.16, 5 000 050 000}
```

We see that it is several times faster. The reasons why this is so, as well as when the procedural programming style (based on loops, conditionals, etc) is and is not appropriate, we will discuss at length in the next chapters. The Timing command that we used gives an elapsed time together with the result of evaluation for a given expression.

To rehabilitate the loops somewhat, let me mention that they can be quite efficient in cases when it is possible to use the internal compilation :

```
Compile[{x, {n, _Integer}},
   Module[{sm = x}, Do[sm = sm + i, {i, n}]; sm]][0, 100 000] // Timing
```

{0.03, 5.00005×10^9}

The numbers returned by compiled functions can however be only machine - size numbers. Also, the compiler can not handle programs which operate on generic objects.

2.9 Four types of brackets in *Mathematica*

Mathematica utilizes four different types of brackets: parentheses, single square brackets, double square brackets and the curly braces. Each type of brackets has a different meaning and purpose.

2.9.1 Parentheses ()

Parentheses are used to change the priority or precedence of different operations, like in other programming languages.

There is another use of parentheses - to group together a block of operators which we want to be considered as a single composite operator (in C curly braces are used for this purpose) - in this case, parentheses are the short hand notation for a CompoundExpression command.

There is no confusion between these two uses of parentheses, since inside the block the operators have to be separated by semicolons, as we already discussed.

2.9.2 Curly braces {}

Curly braces always mean a list. They are equivalent to a literal command List:

```
Clear[a, b, c];
{a, b, c} === List[a, b, c]
True
```

Lists represent collections of objects, possibly of different type, and are very important building blocks of *Mathematica* programs, since any complex data structure can be built out of them. We will cover them in a separate chapter devoted entirely to them.

Some built-in functions use lists of one, two, three or four numbers as interators. One such function - the loop Do - we already encountered.

2.9.3 Single square brackets []

Single square brackets are used as building blocks for normal expressions in *Mathematica*. They are used to represent an arbitrary *Mathematica* expression as a nested tree-like structure, as we already discussed. This is their only (but very important) purpose in *Mathematica*.

You may object that another purpose of them is to be used in function calls, like:

```
Sin[Pi]
0
```

However, the truth is that the notion of function is secondary in *Mathematica*, the primary ones being notions of rules and patterns. Thus, all function calls are at the end (syntactically) just special cases of normal *Mathematica* expressions. The reason that the functions are computed is related to evaluation process and the presence in the global rule base of certain rules associated with the name of the function, but not to some special syntax of function calls (there is no such). For example, the following expression evaluates to itself:

```
Sin[5]
Sin[5]
```

just because there is no rule in the global rule base associated with Sin[5].

■ 2.9.4 Double square brackets [[ ]]

Finally, double square brackets are used as a shorthand notation for a Part command, which is used to deconstruct an expression and index into sub-expressions, extracting them. We have already seen an example of how this can be done.

Warning: a common mistake

It is a very frequent mistake when single square brackets are used in order to extract elements of an array (list) or a matrix (list of lists), for example:

```
Clear[lst];
 lst = {1, 2, 3, 4, 5};
 lst[1]
{1, 2, 3, 4, 5}[1]
```

Here is the correct syntax:

```
lst[[1]]
1
```

The problem is made worse by the fact that the wrong syntax above does not result in an error - this is a legitimate *Mathematica* expression. It just has nothing to do with the intended operation.

```
Clear[lst];
```

■ Summary

We have considered the most elementary operations with variables, such as assignments and comparisons, and also the control structures such as conditionals and loops, which are similar to those found in other procedural languages such as C. The reason that our exposition was rather brief is partly because many of these constructs are rarely used in serious *Mathematica* programming, being substituted by more powerful idioms which we will cover at length in the later chapters.

I also used a few examples to illustrate some more subtle issues related to the rule-based nature of *Mathematica*. These issues are important to understand to get a "feel" of the differences between *Mathematica* and more traditional computing environments.

Along the way, we saw some examples of use of the following built-in functions: Clear, Set, SetDelayed, Equal, SameQ, If, Do, For, While, Apply, Range, Timing.

III. Lists

3.1 Introduction

Lists are the main data structure in *Mathematica*, and also in functional programming languages such as LISP. Any complex data structure can be represented as some (perhaps, complex and nested) list. For example, N-dimensional array is represented as a list with depth N. Any tree can also be represented as a list.

Lists can be generated dynamically during the process of program execution, and the correctly written functions work on lists of arbitrary length without taking the length of the list as an explicit parameter. This results in a quite "clean" code, which is at the same time easy to write. Another advantage of lists is that it is usually easy to debug functions that work on them - we will see many examples of these features. In this chapter we will cover several built-in functions which are used to generate and process lists.

3.2 The main rule of thumb when working with lists in Mathematica

When we work with lists in *Mathematica*, especially large ones, it makes sense to stick to the following main rule: any list we have to treat as a single unit, and avoid operations that break it into pieces, such as array indexing. In other words, the best programming style is to try writing the programs such that the operations are applied to the list as a whole. This approach is especially important in the functional programming style, and leads to significant increase of code efficiency. We will illustrate this issue on many examples, particularly in the chapter V on functional programming.

3.3 The content of lists

Lists are not required to contain only elements of the same type - they can contain any *Mathematica* expressions mixed in an arbitrary way. These expressions may themselves be lists or more general *Mathematica* expressions, of any size and depth. Essentially, a list is a particular case of a general *Mathematica* expression, characterized by having the head List:

```
Clear[x];
List[Sin[x], Cos[x], Tan[x]]
{Sin[x], Cos[x], Tan[x]}
```

3.4 Generation of lists

There are many ways to generate a list

■ 3.4.1 Generating a list by hand

First of all, it is of course possible to generate a list by hand. That is, some fixed lists in the programs are just defined by the programmer. For instance:

```
Clear[testlist, a, b, c, d, e];
testlist = {a, b, c, d, e}
Clear[testlist];

{a, b, c, d, e}
```

3.4.2 Generation of lists of equidistant numbers by the Range command

In practice it is very often necessary to generate lists of equidistant numbers. This is especially true in the functional approach since there functions on such lists replace loops. Such lists can be generated by using the built-in Range command. Examples:

```
Range[5]
Range[2, 10]
Range[2, 11, 3]
Range[0, 1, 0.1]


{1, 2, 3, 4, 5}

{2, 3, 4, 5, 6, 7, 8, 9, 10}

{2, 5, 8, 11}

{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.}
```

3.4.3 Generation of lists with the Table command

In cases when we need lists of more general nature, often they can be generated by the Table command. Examples:

```
Table[1, {i, 1, 10}]
 Table[i^2, {i, 1, 10}]
 Table[i*j, {i, 1, 3}, {j, 1, 3}]
 Table[i + j, {i, 1, 4}, {j, 1, i}]
 Table[i + j + k, {i, 1, 2}, {j, 1, 3}, {k, 1, 3}]
 Table[Sin[i], {i, 1, 10}]

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}

{{1, 2, 3}, {2, 4, 6}, {3, 6, 9}}

{{2}, {3, 4}, {4, 5, 6}, {5, 6, 7, 8}}

{{{3, 4, 5}, {4, 5, 6}, {5, 6, 7}}, {{4, 5, 6}, {5, 6, 7}, {6, 7, 8}}}

{Sin[1], Sin[2], Sin[3], Sin[4],
 Sin[5], Sin[6], Sin[7], Sin[8], Sin[9], Sin[10]}
```

As these examples show, Table accepts one or more lists which indicate the iteration bounds, but we can fill the lists with some functions computed on the counters being iterated. In cases when we have more

As these examples show, Table accepts one or more lists which indicate the iteration bounds, but we can fill the lists with some functions computed on the counters being iterated. In cases when we have more than one iterator, we create a nested list where the innermost iterators correspond to the outermost lists. As we see, the bounds of the more outermost iterators may depend on the variables of more innermost ones, in which case the lists will have sublists of different lengths. This is where we start seeing that lists are more general than (multidimensional) arrays since the sublists are not bound to have the same dimensions. Also, notice that lists created by Table are not bound to be lists of numbers - they can be lists of functions:

```
Clear[i, x];
Table[x^i, {i, 1, 10}]
```

$\{x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}\}$

Here, for example, we created a 3x3 matrix of monomials:

```
Clear[i, j, x];
Table[x^(i+j), {i, 1, 3}, {j, 1, 3}]
```

$\{\{x^2, x^3, x^4\}, \{x^3, x^4, x^5\}, \{x^4, x^5, x^6\}\}$

One more comment about Table is that it is a scoping construct in the sense that it localizes its iterator variables. It effectively uses Block[] scoping construct in doing so, with all the consequences which normally accompany the use of Block[] (see section 4.8). In particular, naively we expect < f[i] > symbol printed 5 times in the following example, since the definition of < f > used the global < i >

```
Clear[f, i];
f := i^2;
Table[f, {i, 5}]
```

```
{1, 4, 9, 16, 25}
```

The global < i > however did not receive any value :

```
i
```

```
i
```

The final comment about Table is that while it is a scoping construct, Return[] command can not be used to break out of it, unlike some other scoping constructs we will encounter :

```
Table[If[i > 3, Return[], i], {i, 10}]
```

```
{1, 2, 3, Return[], Return[], Return[],
 Return[], Return[], Return[], Return[]}
```

■ 3.4.4 A comment on universality of Range

As far as the first and the second examples are concerned, we can get the same result also with the Range command. Observe:

```
Range[10]^0
Range[10]^2

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

All other examples can also be done with Range and a small amount of functional programming:

```
(Range[3] * #) & /@ Range[3]

{{1, 2, 3}, {2, 4, 6}, {3, 6, 9}}

Range[# + 1, 2 * #] & /@ Range[4]
{{2}, {3, 4}, {4, 5, 6}, {5, 6, 7, 8}}

Nest[Partition[#, 3, 1] &, Range[3, 8], 2]
{{{3, 4, 5}, {4, 5, 6}, {5, 6, 7}}, {{4, 5, 6}, {5, 6, 7}, {6, 7, 8}}}
Map[Sin, Range[10]]
{Sin[1], Sin[2], Sin[3], Sin[4],
 Sin[5], Sin[6], Sin[7], Sin[8], Sin[9], Sin[10]}

Clear[x];
 x^Range[10]
```

$\{x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}\}$

The above examples may look unclear now. We give them here just to show that one can go a long way with just the Range command, and to clarify the role of Table. In fact, Table can be thought of as an optimized loop - it is usually (much) more efficient to create lists with Table rather than to do that in a loop such as Do, For or While. But in the functional programming, Table is not so often used, unlike Range, and now you can see why - we can always get the same effect, and moreover, very often do it faster.

3.4.5 Generation of lists inside loops

3.4.5.1 List generation with Append or Prepend

Such loop generation is possible and resembles most closely what we usually have in procedural languages. In *Mathematica* usually this is one of the least efficient ways to do this. Also, the Table command is actually a loop, but optimized for list generation and fast. The direct list generation inside a loop requires the commands Append, Prepend, AppendTo or PrependTo, starting with an empty list. Append adds one element at the end of the list, and for reasons which we will explain later is very inefficient in *Mathematica* for large lists. Let us instead illustrate this with an example. Here is a simple list generated with Append:

```
For[testlist = {}; i = 1, i ≤ 10,
   i ++, testlist = Append[testlist, Sin[i]]];
 testlist
```

```
{Sin[1], Sin[2], Sin[3], Sin[4],
 Sin[5], Sin[6], Sin[7], Sin[8], Sin[9], Sin[10]}
```

This is how we do it with Range:

```
Sin[Range[10]]
```

```
{Sin[1], Sin[2], Sin[3], Sin[4],
 Sin[5], Sin[6], Sin[7], Sin[8], Sin[9], Sin[10]}
```

3.4.5.2 A digression: <myTiming> : a useful function to measure small execution times

Here I introduce a (user - defined) utility function < myTiming >, which I will use heavily in various performance measurements throughout the book :

```
Clear[myTiming];
myTiming[x_] := Module[{z = 0, y = 0, timelim = 0.1,
    p, q, iterlist = (Power[10, #] & /@ Range[0, 10]),
    nm =
     If[ToExpression[StringTake[$Version, 1]] < 6, 2, 1
     ]},
   Catch[
     If[(z = Nest[
           First, Timing[(y ++; Do[x, {#}]);], nm]) > timelim,
        Throw[{z, y}]
        ]
       ] & /@ iterlist] /. {p_, q_} :> p / iterlist[[q]]
  ];

Attributes[myTiming] = {HoldAll};
```

The code is a bit involved to explain now, but after reading through the book it may be a good idea to revisit it, since it illustrates many points. Anyway, for now we are just users.

- 3.4.5.3 Efficiency pitfall: constructing a list with Append

Let us compare the elapsed time of such list generation with that of Range, using the function myTiming :

Check now:

```
myTiming[For[testlist = {}; i = 1,
   i < 1000, i ++, testlist = Append[testlist, Sin[i]]];]
```

```
0.19
```

```
myTiming[Sin[Range[1000]];]
```

```
0.0013
```

We see that constructing a list of (symbolic) values of Sin on a thousand first natural numbers is hundred times slower with Append than with Range. But the truth is that the computational complexities are different and the larger is the list, the more overhead is induced by using Append. We can also see how Table will perform:

```
myTiming[Table[Sin[i], {i, 1000}];]
```

```
0.0023
```

We see that it is a little slower than Range.

In addition to being slower, the generation with the loop introduced a global side effect - the variable <testlist>. So, to make the code clean, one will have in this approach to wrap the loop in an additional modularization construct and thus make it even clumsier.

So, in conclusion, I would thoroughly discourage the reader from straightforward list generation inside a loop. First, there are almost always better design solutions which avoid this problem altogether. Second, there exist workarounds to get a linear time performance in doing it, such as using linked lists (to be discussed below), or indexed variables. Finally, starting with version 5, there are special commands Reap and Sow introduced specifically for efficient generation and collection of intermediate results in computations - we will have an entire chapter in the part II devoted to these commands.

3.5 Internal (full) form of lists

Let me emphasize once again that internal form of lists satisfies the general requirement of how normal expressions are built in *Mathematica* (see chapter I). For example, here are simple and nested lists:

```
Clear[testlist, complextestlist];
testlist = Range[10]
complextestlist = Range /@ Range[5]

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}}
```

And here are their full forms:

```
FullForm[testlist]
List[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
FullForm[complextestlist]
List[List[1], List[1, 2], List[1, 2, 3], List[1, 2, 3, 4], List[1, 2, 3, 4, 5]]
```

This shows in particular that list indexing can be performed in the same way as indexing of more general normal *Mathematica* expressions, outlined in chapter I. We will use this fact in the next subsection.

```
Clear[testlist, complextestlist];
```

3.6 Working with lists and their parts

▪ 3.6.1 List indexing and element extraction with the Part command

▪ 3.6.1.1 Simple lists

Consider a simple list:

```
Clear[testlist];
testlist = Range[1, 20, 3]
{1, 4, 7, 10, 13, 16, 19}
```

Say we need to extract its first element. This is accomplished as follows:

```
testlist[[3]]
7
```

Or, which is the same, like this:

```
Part[testlist, 3]
7
```

Now, let us say we need to extract second, fourth and fifth elements. This is how:

```
testlist[[{2, 4, 5}]]
{4, 10, 13}
```

Or, which is the same:

```
Part[testlist, {2, 4, 5}]
{4, 10, 13}
```

List elements can be also counted from the end of the list. In this case, their positions are negative by convention:

```
testlist[[-1]]
19
```

■ 3.6.1.2 Working with nested lists

Consider now a more complex list:

```
complextestlist = Range[5 * #, 5 * # + 4] & /@ Range[5]
{{5, 6, 7, 8, 9}, {10, 11, 12, 13, 14},
 {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}
```

Its elements of level one (we remind that level is the "distance from the stem", see section 1.1.7) are its sublists:

```
complextestlist[[2]]
{10, 11, 12, 13, 14}
complextestlist[[{1, 4, 5}]]
{{5, 6, 7, 8, 9}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}
```

To get to the numbers, we need in this case a 2-number indexing (since all sublists have a depth 1. Should they have depth N, we would need N+1 indexes, and should the depth be different for different sublists, the number of necessary indices would accordingly be different)

```
complextestlist[[1, 1]]
5
```

which means "first element of the first element" (we can view this list as a 2-dimensional array). Notice that the syntax complextestlist[[{1,1}]] will be interpreted as a twice repeated first element of the original list, i.e. twice repeated first sublist:

```
complextestlist[[{1, 1}]]
{{5, 6, 7, 8, 9}, {5, 6, 7, 8, 9}}
```

Everything that was true for simple lists, is also true here:

```
complextestlist[[-1]]
complextestlist[[-1, -1]]

{25, 26, 27, 28, 29}

29

Clear[testlist, complextestlist];
```

3.6.2 Extract

This operator is analogous to the Part operator, with some extra functionality which is sometimes very useful but of no interest for us at the moment. What is important now, is that it can extract several elements at different levels at the same time (Part can also extract several elements, but they have to be at the same level). Also, Extract has a different syntax - to extract an element on the level deeper than the first (and also, every time when we extract more than one element), the address of the element being extracted should be entered as a list of indices:

```
testlist = Range[1, 20, 3];
complextestlist = Range[5 * #, 5 * # + 4] & /@ Range[5]

{{5, 6, 7, 8, 9}, {10, 11, 12, 13, 14},
 {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}


Extract[testlist, 1]
 Extract[complextestlist, 1]
 Extract[complextestlist, {1, 2}]


1

{5, 6, 7, 8, 9}

6

Extract[complextestlist, {{1, 2}, {3}, {4, 5}}]

{6, {15, 16, 17, 18, 19}, 24}
```

3.6.3 Take and Drop

These commands are used to take or drop from a list several elements in a row. For example, here are again our lists:

```
testlist = Range[1, 20, 3]
complextestlist = Range[5 * #, 5 * # + 4] & /@ Range[5]

{1, 4, 7, 10, 13, 16, 19}

{{5, 6, 7, 8, 9}, {10, 11, 12, 13, 14},
 {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}
```

And these are examples of how Take and Drop work

```
Take[testlist, 3]
{1, 4, 7}

Take[testlist, {2, 4}]
{4, 7, 10}

Take[testlist, -3]
{13, 16, 19}

Take[testlist, {-4, -3}]
{10, 13}

Drop[testlist, 3]
{10, 13, 16, 19}

Drop[testlist, {2, 4}]
{1, 13, 16, 19}

Drop[testlist, -3]
{1, 4, 7, 10}

Drop[testlist, {-4, -3}]
{1, 4, 7, 16, 19}
```

Both Take and Drop also have some extended functionality to automate structural operations performed on nested lists, such as extraction of submatrices from matrices. We don't cover them here, but they are described in *Mathematica* Help.

3.6.4 First, Rest, Last and Most

This commands are is in principle redundant, since First[list] is exactly equivalent to list[[1]], Rest[list] is equivalent to Drop[list,1], Last[list] is equivalent to list[[-1]], and Most[list] is equivalent to Drop[list,-1]. However, they can be used for better code readability.

3.6.5 Length

This command returns the length of the list. For instance:

```
testlist = Range[1, 20, 3]
complextestlist = Range[5 * #, 5 * # + 4] & /@ Range[5]
```

```
{1, 4, 7, 10, 13, 16, 19}
```

```
{{5, 6, 7, 8, 9}, {10, 11, 12, 13, 14},
 {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}
```

```
{Length[testlist], Length[complextestlist]}
```

```
{7, 5}
```

If we want to compute the lengths of sublists in <complextestlist>, this can be done as follows:

```
Table[Length[complextestlist[[i]]], {i, 1, Length[complextestlist]}]
```

```
{5, 5, 5, 5, 5}
```

Here, obviously, the index <i> is used to index the sublists, and thus it runs from 1 to the length of the main list, while the Part operation [[i]] extracts the sublists.

■ 3.6.6 Modification of list elements by direct indexing (using Part)

■ 3.6.6.1 Simple uses of Part

If it is necessary to replace some element of the list with a known address with some new expression or value (say, symbol <a>), this can be done directly. Here are our lists:

```
Clear[a];
testlist = Range[1, 20, 3]
complextestlist = Range[5 * #, 5 * # + 4] & /@ Range[5]
```

```
{1, 4, 7, 10, 13, 16, 19}
```

```
{{5, 6, 7, 8, 9}, {10, 11, 12, 13, 14},
 {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}
```

Say, we now want to replace the element with the address {5} with <a>:

```
testlist[[5]] = a;
testlist
```

```
{1, 4, 7, 10, a, 16, 19}
```

Now, let us now in our list <complextestlist> replace a random element in every sublist with <a>:

```
For[i = 1, i ≤ Length[complextestlist], i ++,
  complextestlist[[i, Random[Integer, {1, 5}]]] = a];
```

```
complextestlist
```

```
{{5, 6, 7, 8, a}, {10, 11, a, 13, 14},
 {15, 16, 17, 18, a}, {20, 21, a, 23, 24}, {25, 26, 27, a, 29}}
```

Notice that such modifications as described above are only possible if lists are stored in some variable (in C we would say, that it is an L-value). In particular, this input is a mistake:

```
Range[10][[3]] = a
Set::setps : Range[10] in the part assignment is not a symbol. >>
a
```

Essentially, in all the above examples the Part command ([[ ]]) was used for modification of expressions.

Note also that the modification of lists with Part introduces side effect, since it is the original variable where the list is stored that is modified, not the copy of the list.

3.6.6.2 Use Part to efficiently modify large structures

One more thing I would like to mention here is that Part has a very powerful extended functionality which allows to change many elements at the same time. For example, if I want to change second element in each sublist of < complextestlist > to < b > and fourth to < c >, I can do this in the following way :

```
Part[complextestlist, All, 2] = {b, c, d, e, f};
complextestlist
{{5, b, 7, 8, a}, {10, c, a, 13, 14},
 {15, d, 17, 18, a}, {20, e, a, 23, 24}, {25, f, 27, a, 29}}
```

This turns out to be very handy in cases when many elements have to be changed at once. This is however limited to cases when these parts form some rather regular structures like sub-matrices etc. For more details, see *Mathematica* Help and *Mathematica* Book, also the description on the web site of Ted Ersek (see Appendices for the URL).

3.6.7 ReplacePart

There is another built-in command used to modify list elements - ReplacePart. But there is a big difference between direct modifications by subscripting (Part) used above, and the functionality of the ReplacePart command. Notice that as a result of our manipulations with Part, the original lists has been modified. However, more often in *Mathematica* programs we would modify a list which is not the original list but a copy, so that the original list remains unchanged. This style of programming is arguably cleaner but requires more memory since the original object is copied. Most built-in functions act in this manner. In particular, this is the case with ReplacePart:

```
Clear[a];
testlist = Range[1, 20, 3]
{1, 4, 7, 10, 13, 16, 19}
ReplacePart[testlist, a, 5]
{1, 4, 7, 10, a, 16, 19}
```

But the original list:

```
testlist
{1, 4, 7, 10, 13, 16, 19}
```

remained unchanged. This also means that ReplacePart does not require L-value to operate on an expression:

```
ReplacePart[Range[10], a, 5]
{1, 2, 3, 4, a, 6, 7, 8, 9, 10}
```

No error in this case - since it did not attempt to change the original list but rather changed the copy of it.

The ReplacePart can be used to change more than a single element at a time, also elements on different levels of expression. For example:

```
ReplacePart[Range[10], a, {{2}, {5}, {8}}]
{1, a, 3, 4, a, 6, 7, a, 9, 10}
```

Note that the syntax regarding the list of positions is the same as for the Extract command. ReplacePart is described in detail in *Mathematica* Help.

I have to mention however, that ReplacePart can become quite slow if one wants to change many parts of a large expression at the same time. For a detailed discussion, please see Appendix C.

Neither do I recommend to use ReplacePart in such cases by changing elements one by one (say, inside a loop) rather than all at once. The reason is that since it effectively copies the entire list and then performs changes on the copy, it will in such a sequential approach copy the entire list as many times as is the total number of replacements. This will be terribly inefficient, just as with Append and Prepend operators. In some of these cases one can efficiently use Part to change many elements at once - one such example we will consider in chapter VI (see section 6.5.5.1).

■ 3.6.8 Position

The Position command is used to determine the positions of elements in a list (or more general *Mathematica* expression) which match a given expression, either exactly, or through patterns.

■ 3.6.8.1 Basic uses of Position

In the simplest form, Position has the following format: **Position[list,element]**. Let us give a few examples:

First we initialize our lists:

```
testlist = Range[1, 20, 3]
complextestlist = Range[5 * #, 5 * # + 4] & /@ Range[5]
{1, 4, 7, 10, 13, 16, 19}
{{5, 6, 7, 8, 9}, {10, 11, 12, 13, 14},
 {15, 16, 17, 18, 19}, {20, 21, 22, 23, 24}, {25, 26, 27, 28, 29}}
```

We will now use the simplest version of Position (without patterns) to obtain all positions where the number <4> is found in a list:

```
Position[testlist, 4]
{{2}}
```

This means that number 4 is the second element in <testlist>

```
Position[complextestlist, 12]
{{2, 3}}
```

This means that number 12 is the third element of the second element (sublist) of the list <complextestlist>.

The Position command can be used together with the Extract command, since they use the same position specifications:

```
Extract[complextestlist, Position[complextestlist, 10]]
{10}
```

3.6.8.2 On less trivial uses of Position

This does not look like a big deal, but we may for example wish to extract the entire sublist containing 10. All we have to do is to construct a new position list, with the last element (index) dropped:

```
Map[Most, Position[complextestlist, 10]]
{{2}}
```

The functionality of Map will be discussed much later, but essentially here it will ensure that if the position list contains several positions, the last index will be dropped from all of them. For example, here we define another nested list in which some numbers are found more than once:

```
complextestlist1 = Range /@ Range[6]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

These are all the positions where number 4 is found:

```
plist = Position[complextestlist1, 4]
{{4, 4}, {5, 4}, {6, 4}}
```

If we use these positions in Extract, we just get number 4 extracted 3 times:

```
Extract[complextestlist1, plist]
{4, 4, 4}
```

However, if we want to extract all the sublists containing 4, we need one more step. To obtain the positions of sublists only, we have to delete the last index from each of the sublists in <plist>:

```
Map[Most, plist]
{{4}, {5}, {6}}
```

Now we can extract sublists:

```
Extract[complextestlist1, Map[Most, plist]]
{{1, 2, 3, 4}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

3.6.8.3 Example: extracting sublists containing given element

Now we will write our first "serious" function: it will extract those sublists from a list, which contain a given element.

```
Clear[sublistsWithElement];
sublistsWithElement[main_List, elem_] :=
  Extract[main, Map[Most, Position[main, elem]]];
```

For example, these are the sublists of <complextestlist1> which contain number 5:

```
sublistsWithElement[complextestlist1, 5]
{{1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

I used this opportunity to illustrate several common features of user-defined functions in *Mathematica*. First, the development of the function: it is usually best to start with some simple test example, develop the code in steps like above, and then package everything into a function. Second, we saw that the parameters in the function definition on the l.h.s. contain an underscore. This is a sign that patterns are used in the definition. For now, I will just briefly mention that pattern like <x_> means "any expression" and automatically makes <x> local for the r.h.s., if SetDelayed (:=)was used in the definition (which is normally the case for functions). The pattern x_h means "anything with the head <h>". Thus, the pattern <main_List> above will match on any list, but not on a more general expression whose head is not List. This constitutes a simple type-check.

The other comment about Position function is due here: it is important to remember that while this is an optimized and fast built-in function, it is still a general-purpose one. In particular, if you have a list sorted with respect to some criteria, it will be typically much faster for large lists to search for an element with a version of a binary search, which you can implement in *Mathematica* (and which has a logarithmic complexity) than with the Position function (which has a linear complexity).

▪ 3.6.8.4 More complicated example - sublists with odd number of odd elements

The problem

We will now illustrate the use of Position with patterns on a somewhat less trivial example. Please ignore the pieces of syntax you are not yet familiar with but rather concentrate on the conceptual part and consider this as an illustration. The problem will be to extract from <complextestlist1> all sublists which have an odd number of odd elements. Our solution will go in steps.

```
complextestlist1 = Range /@ Range[6]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

Developing a solution

Step1: Find all positions of all odd numbers:

```
step1list = Position[complextestlist1, _ ? OddQ]
{{1, 1}, {2, 1}, {3, 1}, {3, 3}, {4, 1},
 {4, 3}, {5, 1}, {5, 3}, {5, 5}, {6, 1}, {6, 3}, {6, 5}}
```

In each of the small sublists, as we already know, the first index gives the number of sublist in <compextestlist1>, and the second one gives the index of the given odd element in this sublist.

Step 2: We combine together the addresses which correspond to the same sublist - they have the same first element:

```
step2list = Split[step1list, First[#1] == First[#2] &]

{{{1, 1}}, {{2, 1}}, {{3, 1}, {3, 3}}, {{4, 1}, {4, 3}},
 {{5, 1}, {5, 3}, {5, 5}}, {{6, 1}, {6, 3}, {6, 5}}}
```

The Split is another built-in command which we will cover shortly and whose purpose is to split a list into sublists of the "same" elements, where the notion of "same" can be defined by the user. In particular, in this case we tell Split to consider sublists of indices "same" if they have the same first element. Notice that now they are combined in extra lists.

Step 3: Leave in the lists only the first elements:

```
step3list = Map[First, step2list, {2}]

{{1}, {2}, {3, 3}, {4, 4}, {5, 5, 5}, {6, 6, 6}}
```

Step4: In the above lists, leave only the sublists with odd length (the length of these sublists corresponds to the number of odd elements in the sublists of our original list, with addresses equal to numbers repeated in the sublists above).

```
step4list = Cases[step3list, x_List /; OddQ[Length[x]]]

{{1}, {2}, {5, 5, 5}, {6, 6, 6}}
```

<Cases> is the command used to find the list of all occurrences of some expression or pattern in a larger expression. We will cover it later.

Step5: Replace all sublists by their first elements:

```
step5list = Union[Flatten[step4list]]
{1, 2, 5, 6}
```

<Flatten> makes any list flat, and <Union> removes duplicate elements and sorts the resulting list.

Step 6: Extracting the sublists:

```
complextestlist1[[step5list]]

{{1}, {1, 2}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

Assembling the code into a function

We can compress all the steps into a single function:

```
Clear[oddSublists];
oddSublists[x_List] :=
 Part[x, Union[Flatten[Cases[Map[First, Split[Position[x, _ ? OddQ],
       First[#1] == First[#2] &], {2}], y_List /; OddQ[Length[y]]]]]]
```

Check:

```
oddSublists[complextestlist1]
{{1}, {1, 2}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

An alternative functional implementation

There is a much simpler but less obvious way to do the same thing by using the mixed rule-based and functional programming style. I will give here the code just for an illustration:

```
Clear[oddSublistsNew];
oddSublistsNew[x_List] :=
  Map[If[EvenQ[Count[#, _?OddQ]], # /. # → Sequence[], #] &, x];
```

Check:

```
oddSublistsNew[complextestlist1]
{{1}, {1, 2}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

While the first realization became significantly complex to question the advantage of this programming style compared to a traditional procedural programming based on nested loops, my primary goal here was to illustrate the use of Position command, and perhaps give a flavor of a few others.

However, the second realization is clearly shorter. This kind of programs can be written very quickly and are typically very short.

A procedural version

It is also less bug-prone than the possible procedural implementation based on 2 nested loops:

```
Clear[oddSublistsProc];
oddSublistsProc[x_List] := Module[{pos = {}, ctr, i, j},
   For[i = 1, i ≤ Length[x], i ++,
    For[j = 1; ctr = 0, j ≤ Length[x[[i]]],
     j ++, If[OddQ[x[[i, j]]], ctr ++];];
    If[OddQ[ctr], AppendTo[pos, i]];];
   Return[x[[pos]]]];
```

Check:

```
oddSublistsProc[complextestlist1]
{{1}, {1, 2}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

Apart from being clumsier, this code also uses AppendTo to append elements to a list, which will make it inefficient for large lists, just as in the examples we considered before.

```
Clear[complextestlist1, step1list, step2list, step3list, step4list,
  step5list, oddSublists, oddSublistsNew, oddSublistsProc];
```

3.7 Adding elements to the list and removing them from the list

3.7.1 Append, Prepend, AppendTo and PrependTo

Some of these commands we have already encountered before. They add an element to the end or to the beginning of the list. For example:

```
Clear[a];
testlist = Range[5]
```

```
{1, 2, 3, 4, 5}
```

```
Append[testlist, a]
```

```
{1, 2, 3, 4, 5, a}
```

```
Prepend[testlist, a]
```

```
{a, 1, 2, 3, 4, 5}
```

```
testlist
```

```
{1, 2, 3, 4, 5}
```

The last output shows that the list <testlist> did not change. As we discussed, the absence of side-effects is typical for *Mathematica* built-in functions. In this case, Append and Prepend forged a copy of <testlist> and modified this copy. If we want an original list to be modified, we have either to write:

```
testlist = Append[testlist, a];
 testlist
```

```
{1, 2, 3, 4, 5, a}
```

or, which is equivalent, to use the function AppendTo, which does exactly this:

```
testlist = Range[5]
```

```
{1, 2, 3, 4, 5}
```

```
AppendTo[testlist, a];
testlist
```

```
{1, 2, 3, 4, 5, a}
```

The situation with Prepend and PrependTo is completely analogous. And also, recalling the previous discussions, we may suspect that the application of AppendTo or PrependTo to a list which is not assigned to any variable (not an L-value) is a mistake, and we will be correct:

```
Append[Range[5], a]
```

```
{1, 2, 3, 4, 5, a}
```

```
AppendTo[Range[5], a]
Set::write : Tag Range in Range[5] is Protected. More…
{1, 2, 3, 4, 5, a}
```

As we already discussed, it is best to avoid using these functions (Append etc) for modifying large lists. Later we will consider several more efficient alternatives.

```
Clear[testlist];
```

■ 3.7.2 Insert and Delete

As is clear from their names, these functions are used to insert an element to or delete it from the list, or more general *Mathematica* expression. These operations are well described in *Mathematica* Help. We will give just a few examples of their use. The format of Insert is **Insert[list,new,pos]** - this will insert the new element <new> in the position <pos> in the list <list>. Delete has a similar syntax **Delete[list,pos]** and respectively deletes from the list <list> an element at position <pos>. For example:

```
Clear[a];
testlist = Range[3, 15, 2]
{3, 5, 7, 9, 11, 13, 15}
Delete[testlist, 4]
{3, 5, 7, 11, 13, 15}
Insert[testlist, a, 5]
{3, 5, 7, 9, a, 11, 13, 15}
```

Notice that once again, both of these commands work on the copy of the original list that they create, so that the original list remains unchanged:

```
testlist
{3, 5, 7, 9, 11, 13, 15}
```

Both of these commands can work on nested lists (or more general expressions), and then the position will be a list of indexes. Also, they may also receive a list of positions rather than a single position - in this case, an element will be inserted or deleted in many places at once.

However, in the case of Insert, it may become quite slow if a large number of parts have to be inserted at the same time. For a more detailed discussion, please see the Appendix C.

3.8 Working with nested lists

It is often necessary to work with nested lists, that is - lists whose elements themselves are lists. We have seen simple examples of such lists already. Let me emphasize that in general such lists are not identical to multidimensional arrays but in fact much more general, because the lengths of sublists at each level can be different. The only thing we can say about the general nested list is that it represents some tree.

Here we will consider several special-purpose commands which were designed for efficient processing of some special types of such nested lists.

3.8.1 Partition

This command is used to "cut" or "slice" some list into (possibly overlapping) pieces. In its simplest form, it has a format **Partition[list, size, shift]**. It cuts the list into pieces with the length <size>, and shifted one with respect to another by <shift>. If the <shift> parameter is not given, the list is cut into non-overlapping pieces. For example:

3.8.1.1 A simple example

```
testlist = Table[Sqrt[i], {i, 1, 10}]
```

$\{1, \sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}\}$

```
Partition[testlist, 3]
```

$\{\{1, \sqrt{2}, \sqrt{3}\}, \{2, \sqrt{5}, \sqrt{6}\}, \{\sqrt{7}, 2\sqrt{2}, 3\}\}$

```
Partition[testlist, 7]
```

$\{\{1, \sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}\}\}$

In the last example, the remaining piece had a size smaller than 7, so it was "eaten up". Now we will partition with overlaps:

```
Partition[testlist, 7, 1]
```

$\{\{1, \sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}\}, \{\sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}\},$
$\{\sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3\}, \{2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}\}\}$

3.8.1.2 An example of practical use: computation of the moving average in a list.

This example is based on a similar discussion in Wagner' 96.

The problem

The m-moving average for a list is an average which is obtained by averaging every element in a list with <m> neighbors to the right and to the left (which means that this quantity is only defined for points

The m-moving average for a list is an average which is obtained by averaging every element in a list with <m> neighbors to the right and to the left (which means that this quantity is only defined for points (elements) having at least m neighbors both to the left and to the right). Thus, moving average is actually a list of such averages, of the length <len>-2m, where <len> is a length of an initial list.

Developing a solution

To solve our problem, we will first define an auxiliary function which will count the average of a list of numbers. However, it will turn out that our function will also work on a list of lists of numbers, this time summing entire lists (with the same number of elements) together, which we will use. So:

```
Clear[listAverage];
listAverage[x_List] := Apply[Plus, x] / Length[x];
```

The expression Apply[Plus,x] computes the sum of elements in the list and its meaning will be explained in chapter V.

Now we will define another auxiliary function:

```
Clear[neighborLists];
neighborLists[x_List, m_Integer] :=
  Partition[x, Length[x] -2*m, 1];
```

For example:

```
neighborLists[testlist, 1]
```

$$\left\{\left\{1, \sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}\right\}, \left\{\sqrt{2}, \sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3\right\}, \left\{\sqrt{3}, 2, \sqrt{5}, \sqrt{6}, \sqrt{7}, 2\sqrt{2}, 3, \sqrt{10}\right\}\right\}$$

Let us now realize that the middle list represents a list of "middle points", and the first and the last list represent here lists of closest "neighbors" for these middle points. Thus, the only thing left to do is to use listAverage on this result:

```
listAverage[neighborLists[testlist, 1]]
```

$$\left\{\frac{1}{3}\left(1+\sqrt{2}+\sqrt{3}\right), \frac{1}{3}\left(2+\sqrt{2}+\sqrt{3}\right), \frac{1}{3}\left(2+\sqrt{3}+\sqrt{5}\right), \frac{1}{3}\left(2+\sqrt{5}+\sqrt{6}\right), \frac{1}{3}\left(\sqrt{5}+\sqrt{6}+\sqrt{7}\right), \frac{1}{3}\left(2\sqrt{2}+\sqrt{6}+\sqrt{7}\right), \frac{1}{3}\left(3+2\sqrt{2}+\sqrt{7}\right), \frac{1}{3}\left(3+2\sqrt{2}+\sqrt{10}\right)\right\}$$

Packaging code to a function

Thus, our final function <movingAverage>will look like:

```
Clear[movingAverage, neighborLists, listAverage];
neighborLists[x_List, m_Integer] :=
  Partition[x, Length[x] -2 * m, 1];
listAverage[x_List] := Apply[Plus, x] / Length[x];
movingAverage[x_List, m_Integer] :=
  listAverage[neighborLists[x, m]];
```

For example, here we find the moving average with two neighbors on each side:

```
movingAverage[testlist, 2]
```

$$\left\{\frac{1}{5}\left(3+\sqrt{2}+\sqrt{3}+\sqrt{5}\right),\ \frac{1}{5}\left(2+\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{6}\right),\ \frac{1}{5}\left(2+\sqrt{3}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(2+2\sqrt{2}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(3+2\sqrt{2}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(3+2\sqrt{2}+\sqrt{6}+\sqrt{7}+\sqrt{10}\right)\right\}$$

Using functional programming to eliminate auxiliary functions

With the help of the functional programming syntax, we can write this as a single function and eliminate the need in auxiliary functions altogether:

```
Clear[movingAverage];
movingAverage[x_List, m_Integer] :=
  (Plus @@ #) / Length[#] &@Partition[x, Length[x] -2 * m, 1];
```

Check:

```
movingAverage[testlist, 2]
```

$$\left\{\frac{1}{5}\left(3+\sqrt{2}+\sqrt{3}+\sqrt{5}\right),\ \frac{1}{5}\left(2+\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{6}\right),\ \frac{1}{5}\left(2+\sqrt{3}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(2+2\sqrt{2}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(3+2\sqrt{2}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(3+2\sqrt{2}+\sqrt{6}+\sqrt{7}+\sqrt{10}\right)\right\}$$

A procedural version

Here is the procedural implementation of the same thing:

```
movingAverageProc[x_List, m_Integer] :=
  Module[{i, j, ln = Length[x], aver, sum},
   aver = Table[0, {ln - 2 * m}];
   For[i = m + 1, i <= ln - m, i ++,
    sum = 0;
    For[j = i - m, j ≤ i + m, j ++,
     sum = sum + x[[j]]];
    aver[[i - m]] = sum / (2 * m + 1)];
   aver];
```

Check:

```
movingAverageProc[testlist, 2]
```

$$\left\{\frac{1}{5}\left(3+\sqrt{2}+\sqrt{3}+\sqrt{5}\right),\ \frac{1}{5}\left(2+\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{6}\right),\ \frac{1}{5}\left(2+\sqrt{3}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(2+2\sqrt{2}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(3+2\sqrt{2}+\sqrt{5}+\sqrt{6}+\sqrt{7}\right),\ \frac{1}{5}\left(3+2\sqrt{2}+\sqrt{6}+\sqrt{7}+\sqrt{10}\right)\right\}$$

Efficiency comparison

The problem with the procedural version is not just that the code is longer, but also that it is more error prone (array bounds, initialization of variables etc). On top of that, it turns out to be far less efficient. Let us compare the efficiency on large lists:

```
Timing[movingAverage[Range[10 000], 10];]
{0.016 Second, Null}
Timing[movingAverageProc[Range[10 000], 10];]
{1.172 Second, Null}
```

Here we have a 100 times difference (for this length of the list)! And moreover, this is not a constant factor, but the difference will increase further with the length of the list. Of course, in procedural languages such as C the latter implementation is natural and fast. Not so in *Mathematica*. However, one can still obtain the code which will be concise, fast and elegant at the same time, with the use of functional programming methods.

```
Clear[testlist];
```

3.8.2 Transpose

This is one of the most useful commands. It has this name since for matrices, which are represented as 2-dimensional lists of lists, it performs the transposition operation. However, we are not forced to always interpret the two-dimensional array as a matrix, especially if it is combined from elements of different types. Then it turns out that the number of useful things one can do with Transpose is much larger. But let us start with the numeric lists: say we have a given list of lists of some elements (they may be lists themselves, but this does not matter for us):

3.8.2.1 Simple example: transposing a simple matrix

```
testlist = Table[i + j, {i, 1, 2}, {j, 1, 3}]
{{2, 3, 4}, {3, 4, 5}}
```

Then,

```
Transpose[testlist]
{{2, 3}, {3, 4}, {4, 5}}
```

3.8.2.2 Example: transposing a matrix of lists

Another example:

```
testlist = Table[{i, j}, {i, 1, 2}, {j, 1, 3}]
{{{1, 1}, {1, 2}, {1, 3}}, {{2, 1}, {2, 2}, {2, 3}}}
```

This is a 2-dimensional array of lists.

```
Transpose[testlist]
{{{1, 1}, {2, 1}}, {{1, 2}, {2, 2}}, {{1, 3}, {2, 3}}}
```

3.8.2.3 Example: combining names with grades

Another example: we have results of some exam - the scores - as a first list, and last names of the students as another one. We want to make a single list of entries like {{student1,score1},...}.

```
Clear[names, scores];
names = {"Smith", "Johnson", "Peterson"};
scores = {70, 90, 50};
```

Then we do this:

```
Transpose[{names, scores}]
{{Smith, 70}, {Johnson, 90}, {Peterson, 50}}
```

But we will get most out of Transpose when we get to functional programming, since Transpose is very frequently used there for efficient structure rearrangements. We will see many examples of its use in the chapters that follow.

3.8.3 Flatten

This command is used to destroy the structure of nested lists, since it removes all internal curly braces and transforms any complicated nested list into a flat one. For example:

- 3.8.3.1 Simple example: flattening a nested list

```
testlist = Table[{i, j}, {i, 1, 2}, {j, 1, 3}]

{{{1, 1}, {1, 2}, {1, 3}}, {{2, 1}, {2, 2}, {2, 3}}}
Flatten[testlist]
{1, 1, 1, 2, 1, 3, 2, 1, 2, 2, 2, 3}
```

- 3.8.3.2 Flattening down to a given level

One can make Flatten more "merciful" and selective by instructing it to destroy only braces up to (or, more precisely, down to) a certain level in an expression. The level up to which the "destruction" has to be performed is given to Flatten as an optional second parameter. For instance:

```
Flatten[testlist, 1]

{{1, 1}, {1, 2}, {1, 3}, {2, 1}, {2, 2}, {2, 3}}
```

Example: flattening a nested list level by level

Another example:

```
testlist = Table[{i, j, k}, {i, 1, 2}, {j, 1, 2}, {k, 1, 3}]

{{{{1, 1, 1}, {1, 1, 2}, {1, 1, 3}}, {{1, 2, 1}, {1, 2, 2}, {1, 2, 3}}},
 {{{2, 1, 1}, {2, 1, 2}, {2, 1, 3}}, {{2, 2, 1}, {2, 2, 2}, {2, 2, 3}}}}
Flatten[testlist, 1]

{{{1, 1, 1}, {1, 1, 2}, {1, 1, 3}}, {{1, 2, 1}, {1, 2, 2}, {1, 2, 3}},
 {{2, 1, 1}, {2, 1, 2}, {2, 1, 3}}, {{2, 2, 1}, {2, 2, 2}, {2, 2, 3}}}
Flatten[testlist, 2]
{{1, 1, 1}, {1, 1, 2}, {1, 1, 3}, {1, 2, 1}, {1, 2, 2}, {1, 2, 3},
 {2, 1, 1}, {2, 1, 2}, {2, 1, 3}, {2, 2, 1}, {2, 2, 2}, {2, 2, 3}}
Flatten[testlist, 3]
{1, 1, 1, 1, 1, 2, 1, 1, 3, 1, 2, 1, 1, 2, 2, 1, 2,
 3, 2, 1, 1, 2, 1, 2, 2, 1, 3, 2, 2, 1, 2, 2, 2, 2, 2, 3}
```

In practice, most frequently one uses either Flatten[expr] to get a completely flat list, or Flatten[expr,1] to remove some internal curly braces which were needed at some intermediate steps but not anymore.

3.8.3.3 Application: a computation of quadratic norm of a tensor of arbitrary rank (vector, matrix etc).

The problem and the solution

Here we will show how the use of Flatten can dramatically simplify the computation of the norm of a tensor of arbitrary rank. What may be surprising is that we will not need the rank of the tensor as a separate parameter. So, we wil start with the code:

```
Clear[tensorNorm];
tensorNorm[x_List] := Sqrt[Plus @@ Flatten[x^2]];
```

It turns out that this tiny code is all what is needed to solve the problem in all generality.

Code dissection

Let us use an example to show how it works. This will be our test matrix:

```
testmatr = Table[i + j, {i, 1, 3}, {j, 1, 3}]


{{2, 3, 4}, {3, 4, 5}, {4, 5, 6}}
```

The norm is the square root of sum of the squares of all matrix elements. First, we will use the fact that arithmetic operations such as raising to some power, can be used on entire lists, because they are automatically threaded over the elements of the list (functions which have this property are said to be Listable). Thus, we first square all the elements:

```
testmatr^2
{{4, 9, 16}, {9, 16, 25}, {16, 25, 36}}
```

Since we don’t care which elements are where but just need to sum them all, we will use Flatten to remove the internal curly braces:

```
Flatten[testmatr^2]
{4, 9, 16, 9, 16, 25, 16, 25, 36}
```

Now we have to some all the elements, and as we saw already this can be done for instance with Plus@@ construction:

```
Plus @@ Flatten[testmatr^2]
156
```

Finally, we have to take a square root:

```
Sqrt[Plus @@ Flatten[testmatr^2]]
```

$2\sqrt{39}$

And we arrive at the code of our function. We see that the function works well on a tensor of any rank without modifications! It would be hard to do this without Flatten, and in particular, in languages like C we would need nested loops to accomplish this (in C, there is also a technique called flattening an array, which consists in exploiting the row-major order in which it is stored in memory and going through the multidimensional array with just a pointer to an integer (or whatever is the type of the smallest array element). Although it usually works, it will be illegal if one wants to strictly adhere to the C standard).

```
Clear[tensorNorm, testmatr];
```

- 3.8.3.4 Application - (relatively) fast list generation with Flatten

As we already mentioned, generating lists straightforwardly in loops is perhaps the worst way to do it, in terms of efficiency. One can use Flatten to speed-up this process considerable. Say, we want to generate a list from 1 to 10 (which is easiest to do, of course, by just using Range[10]). We can do it in the following fashion:

Step 1. We generate a nested list (this type of lists are also called linked lists in *Mathematica*):

```
For[testlist = {}; i = 1, i ≤ 10, i ++, testlist = {testlist, i}];
testlist


{{{{{{{{{{{}, 1}, 2}, 3}, 4}, 5}, 6}, 7}, 8}, 9}, 10}
```

Step 2. We use Flatten:

```
Flatten[testlist]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Let us compare the execution time with the realization with Append described previously:

```
For[testlist = {}; i = 1, i < 5000, i ++, AppendTo[testlist, i]]; //
 Timing
{0.25 Second, Null}
```

Now, with our new method:

```
(For[testlist = {}; i = 1, i < 5000, i ++, testlist = {testlist, i}];
  Flatten[testlist];) // Timing
{0.016 Second, Null}
```

We see that the difference is about an order of magnitude at least. While even this method by itself is not the most efficient, we will later see how linked lists can be used in certain problems to dramatically improve efficiency.

```
Clear[testlist];
```

3.9 Working with several lists

It is often necessary to obtain unions, intersections, and complements of two or more lists, and also to remove duplicate elements from a list. This is done by such built-in commands as Join, Intersection, Complement and Union.

■ 3.9.1 The Join command

The Join command joins together two or several lists. The format of it is **Join[list1,...,listn]**, where <list1,..., listn> are lists, not necessarily of the same depth or structure. If the lists contain identical elements, the elements are not deleted - i.e., the lists are joined together as is, with no further modification of their internal structure. Examples:

```
Clear[a, b, c, d, e, f];

Join[{a, b, c}, {d, e, f}]
{a, b, c, d, e, f}
Join[{{a, b}, {c, d}}, {e, f}]
{{a, b}, {c, d}, e, f}
```

Join connects lists together from left to right, as they are, without any sorting or permuting the elements.

■ 3.9.2 The Intersection command

The Intersection command finds the intersection of two or more lists, that is a set of all elements which belong to all of the intersected lists. The format of the command : **Intersection[list1, ..., listn]**. Examples:

```
Clear[a, b, c, d, e, f];

Intersection[{a, b, c}, {b, c, d, e}]
{b, c}
Intersection[{a, b, c, d, e, f}, {a, b, c}, {c, d, e}]
{c}
Intersection[{a, b, c}, {d, e, f}]
{}
```

In the latter case we got an empty list, because the intersection of the latter two lists is empty.

The Intersection command has an option **SameTest** which can be used to provide a custom "sameness" function - this way we can define our notion of "same" different from the default one. Please see the Union command for an example of use of this option. Also, with this option, Intersection may be slower or much slower than in its "pure" form. For a more detailed discussion, please see Appendix C.

■ 3.9.3 The Complement command

The command **Complement[listAll,list1,...,listn]** gives a complement of the list <listAll> with respect to the union of all other lists <list1,...,listn>. In other words, it returns all the elements of <listAll> which are not in any of <listk>. Note that Complement sorts the resulting list. Examples:

```
Complement[{a, b, c, d, e, f}, {b, c, e}]
{a, d, f}
Complement[{a, b, c, d, e, f}, {b, c, e}, {d, e, f}]
{a}
Complement[{a, b, c, d, e, f}, {b, c, e}, {d, e, f}, {a, b, c}]
{}
```

The Complement command, like Intersection, has the option SameTest, which allows us to define our own notion of "sameness" of objects. All the comments I made for Intersection with this option, also apply here.

3.10 Functions related to list sorting

Here we will discuss three very useful built-in functions related to list sorting. **Sort** function sorts the list. **Union** function removes duplicate elements from the list and also sorts the result. **Split** function splits the list into sublists of same adjacent elements. For all three functions, one can define the notion of "sameness" different from the default one. Below we give more details and examples of use for every of these functions.

3.10.1 The Sort command

3.10.1.1 Basic Sort

This function is used for list sorting. For example :

```
Clear[a, b, c, d, e, f, g, t];
Sort[{a, d, e, b, g, t, f, d, a, b, f, c}]
{a, a, b, b, c, d, d, e, f, f, g, t}
Sort[{5, 7, 2, 4, 3, 1}]
{1, 2, 3, 4, 5, 7}
```

Sort will sort a list of arbitrary *Mathematica* expressions. By default, the sorting is performed lexicographically for symbols, in ascending order for numbers, by the first elements for lists. In general, this is called a canonical sorting order in *Mathematica* - consult *Mathematica* Help for more information.

For example, here we will sort a nested list of integers:

```
nested =
 Table[Random[Integer, {1, 15}], {5}, {Random[Integer, {3, 10}]}]
{{13, 3, 11, 7}, {15, 15, 14, 11, 15, 14},
 {11, 10, 2}, {11, 12, 9, 11, 1, 4}, {7, 4, 15, 11, 9}}
```

```
Sort[nested]
{{4, 13, 3}, {11, 1, 9, 10, 7}, {13, 6, 15, 10, 12, 7},
 {15, 8, 9, 1, 6, 2}, {13, 10, 9, 4, 7, 3, 15, 14, 8}}
```

We see that the sorting is performed by the first element of the sublists.

- 3.10.1.2 Sort with a user-defined sorting function

As an optional second argument, Sort accepts the comparison function to be used instead of the default one. For example:

```
Sort[{5, 7, 2, 4, 3, 1}, Greater]
{7, 5, 4, 3, 2, 1}
```

We can for instance sort our nested list by the length of a sublist. We first define a sorting function:

```
Clear[sortFunction];
sortFunction[x_List, y_List] := Length[x] < Length[y];
```

And now sort:

```
Sort[nested, sortFunction]


{{4, 13, 3}, {11, 1, 9, 10, 7}, {13, 6, 15, 10, 12, 7},
 {15, 8, 9, 1, 6, 2}, {13, 10, 9, 4, 7, 3, 15, 14, 8}}
```

- 3.10.1.3 A first look at pure functions

Mathematica provides a mechanism to construct and use functions without giving them names or separate definitions, the so called "pure functions" (they are called lambda functions in some other languages). We will cover them systematically later, but this is how the previous sorting would look should we use a pure function:

```
Sort[nested, Length[#1] < Length[#2] &]


{{4, 13, 3}, {11, 1, 9, 10, 7}, {13, 6, 15, 10, 12, 7},
 {15, 8, 9, 1, 6, 2}, {13, 10, 9, 4, 7, 3, 15, 14, 8}}
```

Any function of two variables which always returns True or False, can be a sorting function. It is assumed that it gives True or False depending on which element is considered "larger".

I have to mention also that using Sort with a user - defined sorting function may considerably slow down the Sort function. For a more detailed discussion, please see Appendix C.

- 3.10.1.4 The Ordering command

This command gives a permutation of indices needed to sort an input list. It also exists in both "pure" form and with a user - defined comparison function. It gives more information than just Sort, but in particular one can also sort a list using a combination of Ordering and Part.

For example, here is a list we considered before :

```
listtosort = {a, d, e, b, g, t, f, d, a, b, f, c}
{a, d, e, b, g, t, f, d, a, b, f, c}
Ordering[listtosort]
{1, 9, 4, 10, 12, 2, 8, 3, 7, 11, 5, 6}
listtosort[[Ordering[listtosort]]]
{a, a, b, b, c, d, d, e, f, f, g, t}
```

Ordering is a very useful command, exactly because it provides more information than just Sort, while being as efficient as Sort itself. We will see an example of its use in chapter VI.

■ 3.10.2 The Union command

The command **Union[list]** returns a canonically sorted list of all distinct elements of < list > .

■ 3.10.2.1 Basic Union

In its basic form, Union takes a list as a single argument, and returns sorted unique elements in a list. The sorting is done by a default sorting function in *Mathematica* (this is lexicographic for symbolic expressions, in increasing order for numeric ones, by first element for lists etc). Examples :

```
Union[{a, d, e, b, g, t, f, d, a, b, f, c}]

{a, b, c, d, e, f, g, t}
testlist = Table[Random[Integer, {1, 10}], {15}]
{9, 7, 4, 3, 1, 1, 8, 2, 2, 10, 7, 4, 9, 1, 4}

Union[testlist]
{1, 2, 3, 4, 7, 8, 9, 10}
```

The fact that the Union command sorts the resulting list, is intrinsically related with the algorithm that Union uses. If the elements should not be sorted, one can write a custom union function (we will consider a couple of implementations later, see section 5.2.6.2.5), which however will certainly be slower than the built-in Union.

■ 3.10.2.2 Union with the SameTest option

The Union command has an option **SameTest**, which allows us to give it our own definition of which elements have to be considered same. For example, we may consider elements the same if they are the same modulo 3:

```
Union[testlist, SameTest → (Mod[#1 - #2, 3] == 0 &)]
{1, 2, 3}
```

It should be noted that Union with the SameTest function may perform slower or much slower than the "pure" Union. For more details and discussion, please see the Appendix C.

3.10.3 The Split command

This command is used to split the list into several sublists, so that elements in each sublist are the same. This function can accept the "sameness" function as an optional second argument. It goes through the list and compares adjacent elements, using either the default sameness function, which is SameQ, or the sameness function provided to it. Whenever two adjacent elements are not the same, it groups the just passed group of same elements in a sublist and starts a new sublist.

3.10.3.1 Basic Split

In its basic form, Split takes a list to split, as a single argument, and uses the SameQ predicate for element comparison. For example, here we introduce a list and its sorted version:

```
testlist = Table[Random[Integer, {1, 15}], {20}]
sorted = Sort[testlist]
```

```
{8, 12, 10, 3, 13, 15, 13, 6, 6, 2, 4, 9, 5, 11, 6, 10, 7, 4, 15, 5}
```

```
{2, 3, 4, 4, 5, 5, 6, 6, 6, 7, 8, 9, 10, 10, 11, 12, 13, 13, 15, 15}
```

Because in general the adjacent elements in an unsorted list are different, we see that most sublists here will contain a single element:

```
Split[testlist]
```

```
{{8}, {12}, {10}, {3}, {13}, {15}, {13}, {6, 6},
 {2}, {4}, {9}, {5}, {11}, {6}, {10}, {7}, {4}, {15}, {5}}
```

Not so for a sorted list:

```
Split[sorted]
```

```
{{2}, {3}, {4, 4}, {5, 5}, {6, 6, 6}, {7},
 {8}, {9}, {10, 10}, {11}, {12}, {13, 13}, {15, 15}}
```

3.10.3.2 Split with a user-defined sameness function

We can now define two elements the same if, for example, they have the same remainder under the division by 3. However, before using Split to group such elements together, we will have to sort the list with a different sorting function, so that elements which are the same modulo 3 will be adjacent to each other in a sorted list:

```
mod3sorted = Sort[testlist, Mod[#1, 3] < Mod[#2, 3] &]
```

```
{15, 6, 9, 6, 6, 15, 3, 12, 4, 7, 10, 4, 13, 13, 10, 5, 11, 5, 2, 8}
```

Now we can split this list:

```
Split[mod3sorted, Mod[#1, 3] == Mod[#2, 3] &]
```

```
{{15, 6, 9, 6, 6, 15, 3, 12}, {4, 7, 10, 4, 13, 13, 10}, {5, 11, 5, 2, 8}}
```

Split is a very useful function. Since it performs a single run through the list and only compares adjacent elements, its complexity is linear. Also, because the number of comparisons is equal to the length of the list (minus one), it does not suffer so severely from the performance penalties associated with the use of user-defined sameness functions, which we discussed for the Sort and Union functions.

- 3.10.3.3 Example: run-length encoding

One standard application of Split is a run-length encoding. Given a list of possibly repetitive numbers, this encoding consists of replacing this list with a list of elements like {{num1,freq1},...}, where <freqk> gives the total number of consecutive copies of <numk>. For instance, take the result of our first example: all we need to do is to change each sublist to the form just described, which can be done for example like this:

```
Clear[runEncodeSplit];
runEncodeSplit[spl_List] :=
  Table[{spl[[i, 1]], Length[spl[[i]]]}, {i, Length[spl]}];

Clear[runEncode];
runEncode[x_List] := runEncodeSplit[Split[x]];
```

Check:

```
runEncode[sorted]
```

```
{{2, 1}, {3, 2}, {4, 3}, {5, 1}, {6, 2},
 {8, 2}, {9, 1}, {11, 4}, {12, 2}, {14, 1}, {15, 1}}
```

With the functional programming, we can eliminate the need of an auxiliary function runEncodeSplit:

```
Clear[runEncodeFP];
runEncodeFP[x_List] := Map[{First[#], Length[#]} &, Split[x]];
```

Check:

```
runEncodeFP[testlist]
```

```
{{1, 1}, {2, 3}, {3, 1}, {4, 2},
 {8, 3}, {9, 3}, {10, 4}, {13, 1}, {15, 2}}
```

- 3.10.3.4 Example: computing frequencies of identical list elements

As another related application of Split, we will use it in conjunction with Sort to implement a function which will count frequencies of the identical elements in a list. This is extremely easy to do if we notice that we just have to Sort the original list and the use the <runEncode> function on a sorted list:

```
Clear[frequencies];
frequencies[x_List] := runEncode[Sort[x]];
```

Check:

```
frequencies[testlist]
```

```
{{2, 1}, {3, 2}, {4, 3}, {5, 1}, {6, 2},
 {8, 2}, {9, 1}, {11, 4}, {12, 2}, {14, 1}, {15, 1}}
```

In fact, in essentially the same way the function **Frequencies** is implemented in the **'Statistics'DataManip·ulation** add-on package.

There are many other situations where Split is quite useful - we will give further examples of its use in the subsequent chapters.

```
Clear[testlist, sorted, mod3sorted, listDivide,
  frequencies, runEncodeSplit, runEncode, runEncodeFP];
```

Summary

In this chapter we introduced lists - the main building blocks of data structures in *Mathematica*. We considered various operations on lists such as list generation, element extraction, addition, replacement and deletion, locating elements with certain properties in the list, and also several specialized commands for fast structural operations on one or several lists, as well as those related to sorting of lists. The following built-in functions were considered in detail: Range,Table,Part, Extract, Take, Drop, First, Rest, Most, Last, Position, Length, Append, Prepend, AppendTo, PrependTo, Partition, Transpose, Flatten, Join, Union, Intersection, Complement, Sort, Split.

Armed with these functions, we can already go a long way in solving various problems. However, we need another major component for serious program building - an understanding of functions in *Mathematica*: what they are, how to define them, etc. This is a topic of the next chapter.

IV. Rules, patterns and functions

4.1 Introduction

In this chapter we will introduce the notion of functions in *Mathematica*, and also consider numerous examples of functions. Since the *Mathematica* programming language is to a large extent a functional programming language, functions are the central objects here. Also, since lists are used as universal building blocks for data structures, and any complex data structure can be built with lists and modified "on the fly", the emphasis is shifted more towards functions, as compared say to OO programming languages.

Another important aspect of functions and functional programming in *Mathematica* is that this "layer" of programming is built upon a (in my view, more fundamental in *Mathematica*) rule-based engine. This results in the notion of function which is wider than, and also significantly different from that in most other programming languages. Thus, one will not have a complete grasp of functions in *Mathematica* without the understanding of rules and rule-based programming techniques. We will discuss them here as well.

4.2 Rules and patterns

To better understand functions in *Mathematica*, we need a good understanding of patterns and rule substitution. These two topics are just two facets of a single one, since it is the form of the pattern which determines when (on which class of objects) the associated rule will apply.

4.2.1 Rule, RuleDelayed, Replace and ReplaceAll commands

4.2.1.1 Basic rules and the Rule head (function)

It is very easy to organize a rewrite rule in *Mathematica*. For example, the following rule will replace a to b

```
Clear[a, b, ourrule];
ourrule = a → b
```

```
a → b
```

The literal equivalent to the arrow (which represents the rule) is the Rule command. If we look at the FullForm of < ourrule > variable, we see that Rule is used :

```
FullForm[ourrule]
```

```
Rule[a, b]
```

■ 4.2.1.2 How to make the rule apply: ReplaceAll function

By itself, any rule that we define does not do anything useful. The command Rule by itself is just a named container of the left and right hand sides of the rule. It becomes useful when combined with another command, which actually performs the rule substitution in an expression. This command has a format **ReplaceAll[expr, rules]**, and a shorthand notation: <**expr/.rules**>. There can be more than one rule being applied, in which case they should normally be placed into a simple (not nested) list - we will see such examples later . If the rules are placed in a nested list, *Mathematica* interprets them differently - see *Mathematica* Help for details, and also below.

This is, for instance, how our rule will work on some test expression :

```
Clear[f, a, b];
f[a] /. ourrule
f[b]
```

or, which is the same,

```
f[a] /. a → b
f[b]
```

If we have a more complicated expression, where < a > happens more than once, it will be replaced in all places (when we use /., or ReplaceAll command):

```
Clear[f, g, h];
f[a, g[a, h[a]]] /. a → b

f[b, g[b, h[b]]]
```

■ 4.2.1.3 The order in which ReplaceAll tries rules on parts of expressions

Although this is not immediately obvious, often the rule application starts from the larger expression, and if it matches as a whole, then the subexpressions are not checked for further matches. This is so when the pattern (see discussion on patterns below) looks like h[x_] or similar. For example, in this case:

```
Clear[a, q];
{{{a}}} /. {x_} :> q[x]
q[{{a}}]
```

we will need to apply the rule several times to replace all the list braces with <q> - s :

```
{{{a}}} /. {x_} :> q[x] /. {x_} :> q[x]
q[q[{a}]]
{{{a}}} /. {x_} :> q[x] /. {x_} :> q[x] /. {x_} :> q[x]
q[q[q[a]]]
```

But not in this case - here the pattern is just a symbol List :

```
{{{a}}} /. List → q
q[q[q[a]]]
```

This behavior is rather logical, but in cases when a different order of rule substitution is desired, techniques are available to change it. We will discuss them
later (see section 5.2.4.2).

4.2.1.4 Associativity of rules

As the previous example may have suggested, the application of rules is left - associative, meaning that in the expression < expr /. rule1 /. rule2 > is legitimate, and first the rule (or rules if this is a list of rules, see below) < rule1 > will be applied to < expr >, and then the rule (s) < rule2 > will be applied to the result.

4.2.1.5 Locality of rules

It is very important that the rules like the one above are local. This means that when the rule rewrites an object to which it applies into something else, it changes the copy of it, while the initial object remains unchanged. In particular, in the above example, an expression f[a] taken separately, did not change :

```
f[a]

f[a]
```

This is the main difference between the rule and the function which performs a similar transformation - in the latter case a similar rule is defined globally (which means that it will be automatically tried by the kernel on any expression entered interactively or being otherwise evaluated). Essentially, this is the only fundamental difference between functions and lists of rules in *Mathematica*. For example, we can easily simulate the squaring function by a local rule:

```
Clear[f];
{f[x], f[y], f[elephant], f[3]} /. f[z_] :> z^2
```

$\{x^2, y^2, \text{elephant}^2, 9\}$

4.2.1.6 Delayed rules - the RuleDelayed function

I used this example to introduce two new ideas. First, we now have patterns on the left hand side of the rule - they are used to widen the class of objects to which the rule will apply. Second, we have used a new kind of rule (there are only two, and one we already considered before) - the one which uses the **:>** (colon - greater) sign instead of **->** one. The literal equivalent of this is **RuleDelayed[lhs,rhs]** command:

```
RuleDelayed[a, b]
a :→ b
```

As we can guess by the name, this corresponds to a delayed rule substitution - that is, the r.h.s. of the rule is evaluated only after the rule substitution takes place. Later we will consider cases where Rule or RuleDelayed are more appropriate, in more detail, but in general the situation here is similar with the one with Set and SetDelayed assignment operators. This similarity is not accidental, but once again reflects the fact the assignment operators in *Mathematica* are just operators which create global rules.

4.2.1.7 The difference between Rule and RuleDelayed

To illustrate a difference between Rule and RuleDelayed on one particular example, consider a model problem : given a list of elements, substitute every occurrence of the symbol < a > by a random number. Here is our sample list

```
Clear[sample, a, b, c, d, e, f, g, h];
sample = {d, e, a, c, a, b, f, a, a, e, g, a}
```

```
{d, e, a, c, a, b, f, a, a, e, g, a}
```

Now, here is the rule - based solution using Rule :

```
sample /. a → Random[]
```

```
{d, e, 0.177741, c, 0.177741, b, f, 0.177741, 0.177741, e, g, 0.177741}
```

And here is the same using RuleDelayed :

```
sample /. a :> Random[]
```

```
{d, e, 0.655171, c, 0.432888, b, f, 0.564996, 0.30648, e, g, 0.872856}
```

We see that the numbers are the same in the first case and different in the second. This is because, in the first case, the r.h.s. of the rule has been evaluated before it was applied, once and for all. In the second case, the r.h.s. of the rule was re - evaluated every time the rule was applied. As a variation on the theme, suppose we want to substitute each <a> in the list by a list {a,num}, where <num> will be counting the occurrences of <a> in the list. Here is our attempt with Rule:

```
n = 1;
sample /. a → {a, n ++}
```

```
{d, e, {a, 1}, c, {a, 1}, b, f, {a, 1}, {a, 1}, e, g, {a, 1}}
```

Obviously this did not work as intended. And now with RuleDelayed :

```
n = 1;
sample /. a :> {a, n ++}
```

```
{d, e, {a, 1}, c, {a, 2}, b, f, {a, 3}, {a, 4}, e, g, {a, 5}}
```

```
Clear[sample, n];
```

■ 4.2.2 Rule substitution is not commutative

■ 4.2.2.1 Lists of rules

When we have more than just one rule to be tried on an expression, we place the rules in a list. For example:

```
{a, b, c, d} /. {a → 1, b → 2, d → 4}
```

```
{1, 2, c, 4}
```

For all the rules to be tried on an expression, the list of rules has to be a flat list, that is, all the list elements should have a head Rule or RuleDelayed. Supplying a nested list of rules to Replace or ReplaceAll is not an error, but is interpreted as if we want to try all the sublists of rules separately on several copies of our original expression:

```
{a, b, c, d} /. {{a → 1, b → 2}, {c → 3, d → 4}}
{{{1, 2, c, d}, {a, b, 3, 4}}}
```

As a side note, there is nothing special about lists of rules with respect to lists of other *Mathematica* objects:

```
FullForm[{a → 5, a → 6}]
List[Rule[a, 5], Rule[a, 6]]
```

▪ 4.2.2.1 Non-commutativity

The result of the rule substitution depends in general on the order in which the rules are stored in the list, as the following example illustrates.

```
Clear[a, f];
f[a] /. {a → 5, a → 6}
f[a] /. {a → 6, a → 5}


f[5]
f[6]
```

The reason for this is that once some rule has been applied to a given part of an expression, ReplaceAll goes to the next part of an expression and tries the rules on that next part. But even if we run ReplaceAll several times (there is a special command ReplaceRepeated related to this, which we will discuss shortly), the results will still be generally different for different orderings of rules in a list. This is because once a rule applies to a (part of) expression, this part is generally rewritten so that (some of) the rules in our list of rules which applied before will no longer apply, and vice versa.

In any case, our final conclusion is that the rule application is not commutative, and the order of rules in the rule list does matter in general. For an extreme example of this, we will soon consider a rule-based factorial function, where different rule ordering will result in infinite iteration.

▪ 4.2.3 An interplay between rules and evaluation process

When working in *Mathematica*, it is important to remember that we never actually start from scratch, but always with a large built-in system of rules which connect together the built-in functions. This gives great flexibility in using these functions, since these system rules can be manipulated or overridden with the user-defined ones. On the other hand, one has to be more careful, because the rules (or function definitions and variable assignments, which are global rules) newly defined by the user, immediately start to interact with the built-in ones. The mentioned above non-commutativity of rules can make this interaction quite non-trivial. This often results in some unexpected or "erroneous" behavior, which many *Mathematica* users immediately proclaim as bugs, but which can be avoided just by getting a better understanding of how the system works.

■ 4.2.3.1 When the rule applies, expression is evaluated

As one example, consider a gamma-function of the symbolic argument:

```
Clear[a];
Gamma[a]
```

```
Gamma[a]
```

Since the system does not know what < a > is, no one of the rules associated with the gamma - function applies, and the input is just returned back. Let us now use the following rewrite rule :

```
Gamma[a] /. a → 5
```

```
24
```

We see that as soon as the numerical (integer) value has been substituted, one of the built - in rules applied, producing the result. At the same time, for a number π (for instance), there is no rule which forces *Mathematica* to compute the numerical value, and thus we have :

```
Gamma[a] /. a → Pi
```

```
Gamma[π]
```

If we want to compute a numerical value in this case, we can either do this :

```
Gamma[a] /. a → N[Pi]
```

```
2.28804
```

Or, which is equivalent (with some tiny differences unimportant now), this :

```
N[Gamma[a] /. a → Pi]
```

```
2.28804
```

(the N function computes a numerical value of its argument). What I want to stress is that the decision whether to keep say Gamma[5] as it is here or to substitute it by its numerical (well, integer) value is rather arbitrary in the sense that it is defined by certain (very sensible) *Mathematica* conventions but there is no first principle which tells which form of the answer is advantageous. In fact, in some cases I may wish to keep a Gamma[5] function in its unevaluated form. More generally, the whole advantage in using rule-based approach is that we don't need a first principle to add rules for a new situation that we want to describe.

This means not that *Mathematica* is unpredictable, but that the programs we write should not depend on features that are defined purely by conventions. In particular, in *Mathematica* one should always assume that all expressions may evaluate to something else. Thus, if some expression has to be kept unevaluated for some time, the programmer has to take care of it. On the other hand, if some expression must be evaluated completely (say, to a numerical value), once again the programmer has to ensure it.

4.2.3.2 Evaluation affects applicability of rules

Consider now a different example:

```
{f[Pi], Sin[Pi], Pi^2}
{f[Pi], Sin[Pi], Pi^2} /. Pi → a
```

```
{f[π], 0, π^2}
{f[a], 0, a^2}
```

Note that in the second input in the list, we will naively expect Sin[a] instead of 0 as an output, in the case when we apply the rule Pi -> a. The reason for this result being as it is can be understood easily, once we recall that the sign /. is just an abbreviation, and equivalently we can write the last input as

```
ReplaceAll[{f[Pi], Sin[Pi], Pi^2}, Pi → a]
```

```
{f[a], 0, a^2}
```

Now we recall the general evaluation strategy in *Mathematica*, where the subexpressions are normally evaluated before the expression itself. This means that once the evaluation process reached ReplaceAll command, our expression has been already transformed to the same form as the output of the first input (without rule substitution). Sin[Pi] evaluated to 0, and since 0 does not contain < Pi > any more and thus does not match the rule, no further substitution took place for this part of our expression. Once again, we can see the evaluation dynamics by using the Trace command:

```
Trace[{f[Pi], Sin[Pi], Pi^2} /. Pi → a]
```

```
{{{Sin[π], 0}, {f[π], 0, π^2}}, {f[π], 0, π^2} /. π → a, {f[a], 0, a^2}}
```

These examples may give an impression that *Mathematica* is unstable with respect to bugs related to rule orderings. While it is true that many non-trivial bugs in the *Mathematica* programs are related to this issue, there are also ways to avoid them. As long as the rule or list of rules are always correct in the sense that either they represent exact identities (say, mathematical identities), or one can otherwise be sure that in no actual situation they, taken separately, will lead to an incorrect result, it should be fine.

Bugs happen when rules considered "correct" give incorrect results in certain unforeseen situations, but this is also true for programs written within more traditional programming paradigms. Perhaps, the real difference is that for more traditional programming techniques it is usually easier to restrict the program to "stay" in those "corners" of the problem parameter space where correct performance can be predicted or sometimes proven. I personally view the complications arising due to rule orderings as a (possibly inevitable) price to pay for having a very general approach to evaluation available.

4.2.4 Rules and simple (unrestricted) patterns

Let us give some examples of how rules work with the simplest patterns.

4.2.4.1 A simplest pattern and general pattern-matching strategy

The simplest pattern of all, which we have already seen before, is just a single underscore <_>, and has a literal representation **Blank[]**:

```
Blank[]

_
```

This pattern represents any *Mathematica* expression. Let us take some sample *Mathematica* expression:

```
x^y * Sin[z]
```

Now we will use our simplest pattern to replace any element by, say, a symbol < a >:

```
Clear[a, x, y, z, g, h];
{x, Sin[x], x^2, x * y, x + y, g[y, x], h[x, y, z], Cos[y]} /. _ → a

a
```

This is not very exciting. What happened is that our entire expression (list) matched this pattern and then got replaced by < a > . Before we move forward, let me explain a bit how patterns work and why the substitution based on patterns is possible. The main ingredient for this is the uniform representation of *Mathematica* expressions by symbolic trees. Basically, when we try to match some (however complex) pattern with some expression, we are matching two trees. The tree that represents the pattern is also a legal *Mathematica* expression (patterns are as good *Mathematica* expressions as anything else), but with some branches or leaves replaced by special symbols like Blank[] (the underscore). For example:

```
FullForm[(_^_) * Sin[_]]

Times[Power[Blank[], Blank[]], Sin[Blank[]]]
```

This pattern tree (or, just pattern) will match some expression < expr > if they are identical modulo some parts of < expr > which can be "fit" in the placeholders such as Blank[], present in this pattern. In particular, the pattern above will match any expression which is a product of something to a power of something else, and a Sine of something.

- #### 4.2.4.2 Does the pattern match? The MatchQ function

There is a very useful command that allows one to check whether or not there is a match between a given expression and a given pattern: **MatchQ**. It takes as arguments an expression and a pattern and returns True when pattern matches and False otherwise. For example:

```
MatchQ[x^y * Sin[z], (_^_) * Sin[_]]

True

MatchQ[Exp[-x^2]^2 * Sin[Cos[x - y]^2], (_^_) * Sin[_]]

True

MatchQ[x * Sin[z], (_^_) * Sin[_]]

False
```

It is important to understand that the pattern - matching (for simple, or unrestricted, patterns) is based completely on syntax, and not semantics, of the expressions being matched.

4.2.4.3 Pattern tags(names) and expression destructuring

Now, while there is some value in just establishing the fact that some expression matches certain pattern, it becomes much more useful when we get access to the parts of this expression which match certain parts of the pattern, so that we can further process these parts. This is called expression destructuring, and is a very powerful pattern - related capability. For instance, in the above example we may wish to know which expressions were the base, the power and the argument of Sine. But to be able to do such destructuring, we need to somehow label the parts of the pattern. This is possible through the mechanism of pattern tags (or names) : we attach some symbol to the pattern part, and then this symbol stores the corresponding part of the actual expression matched, ready for further processing. This is how, for example, we can insert tags in our pattern:

```
(base_^pwr_) * Sin[sinarg_]
```

The pattern tags can not be composite expressions - only true symbols (with the head Symbol).

The presence of pattern tags does not change the matching in any way, it just gives us additional information. We will not obtain this information through MatchQ, however, since MatchQ just establishes the fact of the match. We will need a real rule application for that, since the rule will tell us what to do with these matched (sub) expressions. For example, here we will simply collect them in a list :

```
{x^y * Sin[z], Exp[-x^2]^2 * Sin[Cos[x -y]^2]} /.
 (base_^pwr_) * Sin[sinarg_] → {base, pwr, sinarg}
```

$\{\{x, y, z\}, \{e, -2\,x^2, \mathrm{Cos}[x-y]^2\}\}$

What we just did is exactly a special case of destructuring. The parts that we tagged are now available for whatever operations we would like to perform.

So, to summarize: whenever a pattern contains a part which is a special symbol like Blank[] (there are a few more like it, we will cover them shortly), possibly with a pattern tag attached, this means that the actual expression matched can contain in this place a rather arbitrary subexpression (how arbitrary, depends on the particular special symbol used). However, the parts which do not contain these symbols (multiplication, Power and Sin in our example above), have to be present in exactly the same way in both pattern and the expression, for them to match.

One more important point about pattern tags is that the two identical pattern tags (symbols) in different parts of a pattern can not stand for different corresponding subexpressions in the expression we try to match. For example :

```
MatchQ[a^a, b_^b_]


True
```

```
MatchQ[a^c, b_^b_]
False
```

▪ 4.2.4.4 Example: any function of a single fixed argument

The following pattern will work on every function, but of the argument which has to literally be <x>:

```
Clear[f, x];
f_[x]
```

It can be used for instance when we need to substitute the literal < x > by some value (say, 10) at every place where it appears as a single argument of some function. Consider now some list of expressions which we will use throughout this section to illustrate the construction of various patterns :

```
Clear[x, y, z, g, h, a];
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]}
```

$\{x, \mathrm{Sin}[x], x^2, x\,y, x+y, g[y, x], h[x, y, z], \mathrm{Cos}[y]\}$

We now use our pattern :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 {f_[x] → f[10]}
```

$\{x, \mathrm{Sin}[10], x^2, x\,y, x+y, g[y, x], h[x, y, z], \mathrm{Cos}[y]\}$

The replacement happened only in the second element of the list. To understand this, we have to recall the tree - like nature of *Mathematica* expressions and also that the rules application is based on expression syntax rather than semantics. Let us look at the FullForm of these expressions:

```
FullForm[{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]}]
```

```
List[x, Sin[x], Power[x, 2], Times[x, y], Plus[x, y], g[y, x], h[x, y, z], Cos[y]]
```

We see that only the second list element contains < x > as a single argument, the next four have 2 arguments (and thus the pattern does not match), the next one has 3 arguments, and the last one has a single argument, but < y > instead of < x > .

▪ 4.2.4.5 Any function of 2 arguments, but with the first fixed

Let us now develop other rules which will work selectively on different groups of these elements. First, let us build a rule which will work on elements with 2 arguments - this is easy :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 {f_[x, z_] → f[10, z]}
```

$\{x, \mathrm{Sin}[x], 100, 10\,y, 10+y, g[y, x], h[x, y, z], \mathrm{Cos}[y]\}$

We see that a new pattern is f_[x, z_], which means any function of two arguments, with an arbitrary second argument, but the first argument fixed to literal < x > . Notice that no substitution was performed for g[y,x], since here <x> is the second argument, while in the pattern it is the first one. In general, for simple patterns, the way to determine whether or not a given pattern will match a given expression is to consider the FullForm of both the pattern and the expression, and see whether the pattern expression has blanks in all places where it is different from the expression it has to match. The FullForm of the above pattern is

```
FullForm[f_[x, z_]]
Pattern[f, Blank[]][x, Pattern[z, Blank[]]]
```

To get a better idea of how the above pattern matched the expressions in our list, we may construct a different rule, which will show us which parts were matched :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], Cos[y]} /.
 {f_[x, z_] → {{"f now", f}, {"z now", z}}}
{x, Sin[x], {{f now, Power}, {z now, 2}}, {{f now, Times}, {z now, y}},
 {{f now, Plus}, {z now, y}}, g[y, x], Cos[y]}
```

Returning to our previous rule for functions of 2 arguments, let us first fix the problem with the g[y, x] term so that it also matches. Our solution will be to add another rule, thus making a list of replacement rules :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 {f_[x, z_] → f[10, z], f_[z_, x] → f[z, 10]}
{x, Sin[x], 100, 10 y, 10+y, g[y, 10], h[x, y, z], Cos[y]}
```

- 4.2.4.6 Combining 1, 2 and 3-argument cases together

Now, let us change our rules such that in the Sin[x] the replacement will also take place - that is, now we want < x > to be replaced in functions of either one or two arguments. For this, we combine our very first rule with the last two:

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 {f_[x] :→ f[10], f_[x, y_] :→ f[10, y], f_[z_, x] → f[z, 10]}

{x, Sin[10], 100, 10 y, 10+y, g[y, 10], h[x, y, z], Cos[y]}
```

Finally, we would like to add a rule to include a function of 3 variables :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 {f_[x] → f[10], f_[x, y_] → f[10, y],
  f_[z_, x] → f[z, 10], f_[x, y_, z_] → f[10, y, z]}

{x, Sin[10], 100, 10 y, 10+y, g[y, 10], h[10, y, z], Cos[y]}
```

- 4.2.4.7 New pattern - building block - BlankSequence - and a caution

Let me now introduce another pattern - building block similar to Blank[], which simplifies the construction of many patterns. The literal command for this new pattern object is BlankSequence[]. Its literal equivalent is a double underscore sign < __ > :

```
BlankSequence[]

__
```

This pattern element is used to represent a sequence of one or more elements in an expression. For example :

```
{g[x, y, z], g[p, q], h[x, y]} /. g[t__] → {t}
{{x, y, z}, {p, q}, h[x, y]}
```

However, since this pattern - building block allows one to create patterns which match a much wider class of expressions, sometimes we get not what we would like to. Here, for instance, we want to add a literal <a> as an extra argument to all the functions in the list:

```
{g[x, y, z], h[x, y]} /. f_[t__] -> f[t, a]
{g[x, y, z], h[x, y], a}
```

Instead, < a > was added to the list itself. Why this happened is easy to see from the FullForm of our list :

```
FullForm[{g[x, y, z], h[x, y]}]
List[g[x, y, z], h[x, y]]
```

We see that the head List matches < f_ > (which means any head), and t__ is matched by the interior of the list, since it is indeed a sequence of 2 elements - functions < g > and < h > . Thus, < a > has been added as a last argument to our list of functions, instead of being added to each function in the list. We see that our rule applied 1 level higher than we wanted - to the entire expression rather than parts. This is a common situation when ReplaceAll is used. In this case, the rules are first tried on the larger expressions, and if they match, the rules are applied to them, while subexpressions are not tried any more. There are several ways to change this behavior. One way is to create restricted patterns to make them more selective. Another way is to use the Replace command, which performs rule substitution in a different order.

■ 4.2.4.8 First look at conditional (restricted) patterns

If we choose the first way, we have to modify our pattern such that the head List will not match. The thing we would like to do is to impose a condition that < f > in the pattern < f > is not the same as List. This is how it is done (more details on restricted patterns later) :

```
{g[x, y, z], h[x, y]} /. f_[t__] /; f =!= List -> f[t, a]
{g[x, y, z, a], h[x, y, a]}
```

Now we see that it works as intended.

■ 4.2.4.9 Using Replace instead of ReplaceAll

If we choose to use the Replace command, here is a way to do the same :

```
Replace[{g[x, y, z], h[x, y]}, f_[t__] -> f[t, a], 1]
{g[x, y, z, a], h[x, y, a]}
```

The difference between Replace and ReplaceAll is that the former allows us to specify, in which level of expression the rules have to be applied. This gives a more detailed control over the application of rules.

Here, for instance, we required that the rules be applied only at level 1, which means the level of list subexpressions - functions < g > and < h > .

Another difference is that even when we indicate that the rules have to be applied to the whole expression, which we can do by using Replace[expr, rules, Infinity], the order in which the rules will be applied is different from that of ReplaceAll : now the rules will be applied in the depth - first manner, from the smallest (inner) subexpressions to larger (outer) ones. This behavior is often preferred, and then one has to use Replace. We will use this observation in later chapters.

■ 4.2.4.10 Rule within a rule, and a better catchall solution for our example

Considering our initial problem, it is rather inconvenient to have so many rules for describing essentially some particular cases of a general situation : whenever literal < x > is an argument of some function, change it to < 10 > in that function. Let us see if we can write a more concise and general solution for this problem. The one that I suggest requires BlankSequence.

Illustrating expression deconstruction

To illustrate how it works here, we will now create a rule which will "deconstruct" our expressions according to this pattern :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_[t__] /; f =!= List → {{"function", f}, {"args", {t}}}
```

```
{x, {{function, Sin}, {args, {x}}},
 {{function, Power}, {args, {x, 2}}},
 {{function, Times}, {args, {x, y}}},
 {{function, Plus}, {args, {x, y}}}, {{function, g}, {args, {y, x}}},
 {{function, h}, {args, {x, y, z}}}, {{function, Cos}, {args, {y}}}}
```

The reason that I used a restricted pattern with a condition f =!= List is that otherwise the whole list will match, since its elements match the pattern __ :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_[t__] → {{"function", f}, {"args", {t}}}
```

```
{{function, List},
 {args, {x, Sin[x], x^2, x y, x+y, g[y, x], h[x, y, z], Cos[y]}}}
```

Restricted patterns we will cover later, for now just think of it as a pattern with a certain condition attached. At the same time, this example by itself shows that one has to be careful when defining patterns since the can match in more cases than we expect.

The solution to our problem

Returning to our initial problem, here is a solution:

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_[t__] /; f =!= List :> (f[t] /. x → 10)
```

```
{x, Sin[10], 100, 10 y, 10+y, g[y, 10], h[10, y, z], Cos[y]}
```

Explanation

1. Use parentheses to impose the correct precedence

This solution is remarkable in several aspects. First of all, we have used a rule within a rule, and the inner rule we used in parentheses. The meaning of this is : once we found anything that matches the pattern of the "first" (or "outer") rule, make a replacement in this expression according to the "second", or "inner" rule. The first ("outer") rule acts here essentially as a filter for a second ("inner") one. Should we omit the parentheses, and we would not get the desired result :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
  f_[t__] /; f =!= List :> f[t] /. x → 10
```

```
{10, Sin[10], 100, 10 y, 10+y, g[y, 10], h[10, y, z], Cos[y]}
```

The reason this happened is that the rule application is left - associative. Thus, the first rule applied first, and was essentially idle, because it says by itself just f_[t__] /; f =!= List :> f[t], that is, replace an expression of this form by itself :

```
step1 = {x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
  f_[t__] /; f =!= List :> f[t]
```

$\{x, \mathrm{Sin}[x], x^2, x\,y, x+y, g[y, x], h[x, y, z], \mathrm{Cos}[y]\}$

The second rule then applied to < step1 >, and it says : replace < x > by < 10 > whenever < x > occurs. Thus, for example, the first element of the list, which is just < x > and is not an argument of any function, also got replaced.

```
step1 /. x → 10
```

```
{10, Sin[10], 100, 10 y, 10+y, g[y, 10], h[10, y, z], Cos[y]}
```

However, if we use the parentheses, we ensure that the second rule is actually applied to the results "filtered" by the first rule.

2. Use RuleDelayed to supply correct arguments to the inner rule

The second subtlety here is that we used RuleDelayed instead of Rule for the first (outer) rule. It is easy to see that if we don’t , we will not get a desired result :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_[t__] /; f =!= List -> (f[t] /. x → 10)
```

$\{x, \mathrm{Sin}[x], x^2, x\,y, x+y, g[y, x], h[x, y, z], \mathrm{Cos}[y]\}$

The reason is that in the case of Rule, the r.h.s. of the rule is evaluated before the rule is substituted. Recalling the general strategy of *Mathematica* evaluation, the inner rule will be evaluated before the outer one (this is what parentheses ensure). But by that time, the literal < t > in the outer rule has yet no relation to the matched parts of the list whatsoever, and just remains the literal < t > . Thus, the rule < f[t] /. x -> 10 > is completely idle and evaluates to f[t]. This results then in just the first rule f_[t__] /; f =!= List ->f[t], which again is idle.

3. Confirmation : use Trace command

This description can be confirmed with the Trace command:

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
  f_[t__] /; f =!= List -> (f[t] /. x → 10) // Trace
```

```
{{{f[t] /. x → 10, f[t]},
  f_[t__] /; f =!= List → f[t], f_[t__] /; f =!= List → f[t]},
 {x, Sin[x], x^2, x y, x + y, g[y, x], h[x, y, z], Cos[y]} /.
  f_[t__] /; f =!= List → f[t], {List =!= List, False},
 {Sin =!= List, True}, {Power =!= List, True},
 {Times =!= List, True}, {Plus =!= List, True},
 {g =!= List, True}, {h =!= List, True}, {Cos =!= List, True},
 {x, Sin[x], x^2, x y, x + y, g[y, x], h[x, y, z], Cos[y]}}
```

If we use RuleDelayed however, the r.h.s of the outer rule (which includes the inner rule) will not be evaluated until some match has been found for the outer rule. This will allow the inner rule to operate on expressions matched by the outer rule.

■ 4.2.4.11 Rule vs. RuleDelayed - when which one is appropriate

When RuleDelayed is better

In general, it is usually safer to use RuleDelayed in cases when the r.h.s. of the rule is not constant. For instance, all our replacements just discussed would have been totally spoiled had the parts of the pattern definition tags (such as f in f_, t in t__, etc) have any global values. This is, for instance, one of the rules we used above, but with some pattern tags having some global values before the application of rule:

```
f = "Global function";
t = "global value";
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_[t__] /; f =!= List → {{"function", f}, {"args", {t}}}
```

```
{x, {{function, Global function}, {args, {global value}}},
 {{function, Global function}, {args, {global value}}},
 {{function, Global function}, {args, {global value}}},
 {{function, Global function}, {args, {global value}}},
 {{function, Global function}, {args, {global value}}},
 {{function, Global function}, {args, {global value}}},
 {{function, Global function}, {args, {global value}}}}
```

Once again, this is because the r.h.s. of the rule has been evaluated before any match could have taken place. To be safe, use RuleDelayed :

```
f = "Global function";
t = "global value";
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_[t__] /; f =!= List :> {{"function", f}, {"args", {t}}}
```

```
{x, {{function, Sin}, {args, {x}}},
 {{function, Power}, {args, {x, 2}}},
 {{function, Times}, {args, {x, y}}},
 {{function, Plus}, {args, {x, y}}}, {{function, g}, {args, {y, x}}},
 {{function, h}, {args, {x, y, z}}}, {{function, Cos}, {args, {y}}}}
```

```
Clear[f, t];
```

When Rule is better

There are of course cases when Rule has to be used instead - this is when the r.h.s. of the rule is a constant. For example, say we want to substitute every even number in a list by the value of the integral of the gaussian over the half-line. This is how long it takes with Rule (for this size of the list)

```
Range[50] /. _?EvenQ → Integrate[Exp[-x^2], {x, 0, Infinity}] //
  Short // Timing
```

$\left\{0.101, \left\{1, \frac{\sqrt{\pi}}{2}, 3, \ll 44 \gg, \frac{\sqrt{\pi}}{2}, 49, \frac{\sqrt{\pi}}{2}\right\}\right\}$

And now with RuleDelayed :

```
Range[50] /. _?EvenQ :> Integrate[Exp[-x^2], {x, 0, Infinity}] //
  Short // Timing
```

$\left\{2.113, \left\{1, \frac{\sqrt{\pi}}{2}, 3, \ll 44 \gg, \frac{\sqrt{\pi}}{2}, 49, \frac{\sqrt{\pi}}{2}\right\}\right\}$

What happens here is that in one case, the integral is computed exactly once, while in the other it is recomputed over and over again. Once again, the situation here is very similar to the one with immediate and delayed assignments Set and SetDelayed.

- 4.2.4.12 Patterns testing heads of expressions

Let us again return to our original list of expressions. We can also construct more stringent rules which will operate only on certain functions, for example :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_Sin :→ (f /. x → 10)
```

$\{x, \text{Sin}[10], x^2, x\,y, x+y, g[y, x], h[x, y, z], \text{Cos}[y]\}$

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_Plus :→ (f /. x → 10)
```

$\{x, \text{Sin}[x], x^2, x\,y, 10+y, g[y, x], h[x, y, z], \text{Cos}[y]\}$

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 f_Power :> (f /. x -> 10)
```

```
{x, Sin[x], 100, x y, x+y, g[y, x], h[x, y, z], Cos[y]}
```

We have introduced another useful part of the pattern - the part which checks the Head. Although strictly speaking such patterns should be already considered restricted patterns and this topic we did not cover yet, this construction is purely syntactic and thus still appropriate for our present discussion. The idea is that the pattern x_head will match any single expression with the Head < head > .

- 4.2.4.13 On one pitfall of pattern - building

In the first case, by writing the pattern as above, we did not save much:

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
  Sin[x] :> Sin[10];
```

```
{x, Sin[10], x^2, x y, x+y, g[y, x], h[x, y, z], Cos[y]}
```

But in the second case, we have avoided one of the pitfalls associated with the pattern - building :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 Plus[t__] :> (Plus[t] /. x -> 10)
```

```
{10, Sin[10], 100, 10 y, 10+y, g[y, 10], h[10, y, z], Cos[y]}
```

What happened? This kind of behavior creates nightmares for those starting to fool around with rules and patterns. This is generally beyond the scope of the present discussion, but what happened here is that pattern itself evaluated before it had any chance to match in its original form:

```
Plus[t__]
```

```
t__
```

The real pattern that was matched then was not Plus[t__] but just t__, with the above result (RuleDelayed does not evaluate the r.h.s. of the rule, but it of course does evaluate the l.h.s). The solution in such cases is to use the **HoldPattern** command:

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 HoldPattern[Plus[t__]] :> (Plus[t] /. x -> 10)
```

```
{x, Sin[x], x^2, x y, 10+y, g[y, x], h[x, y, z], Cos[y]}
```

Now it works as intended. The same problem is with the Power function, which also evaluates to its argument given a single argument:

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 Power[t__] :> (Power[t] /. x -> 10)
```

```
{10, Sin[10], 100, 10 y, 10+y, g[y, 10], h[10, y, z], Cos[y]}
```

The correct way :

```
{x, Sin[x], x^2, x*y, x+y, g[y, x], h[x, y, z], Cos[y]} /.
 HoldPattern[Power[t__]] :> (Power[t] /. x -> 10)
{x, Sin[x], 100, x y, x+y, g[y, x], h[x, y, z], Cos[y]}
```

The uses of HoldPattern and related things will be covered systematically later. Returning to our current topic, we see that the forms x_head and head[x__] are similar but not entirely equivalent.

- 4.2.4.14 Matching expressions with any number of elements - BlankNullSequence

The other source of inequivalence between these two forms can be seen on the following example :

```
{f[x], g[x], f[x, y], Sin[x+y], f[], f[x, y, z]} /. f[t__] :> a*f[t]
{a f[x], g[x], a f[x, y], Sin[x+y], f[], a f[x, y, z]}
```

Here, we multiplied by < a > all expressions with head < f > . However, the one with no arguments was missed. This does not happen for the <x_f> variant below - here all occurrences of <f> are incorporated, with or without arguments.

```
{f[x], g[x], f[x, y], Sin[x+y], f[], f[x, y, z]} /. x_f :> a*x
{a f[x], g[x], a f[x, y], Sin[x+y], a f[], a f[x, y, z]}
```

Notice by the way that the < x > in the pattern <x_f> has nothing to do with the global <x> (present for example in the expressions in the list), but rather has to be treated as a local variable whose scope is limited to the rule. Other pattern tags are usually local for RuleDelayed but global for Rule, as we already discussed.

To make a structural pattern f[t__] "catch up" with the <x_f> one, we need to introduce another (last in this category) pattern-building block: <BlankNullSequence>, which has a triple underscore as abbreviation:

```
BlankNullSequence[]
___
```

Its purpose is to match a sequence of zero or more expressions. Check now :

```
{f[x], g[x], f[x, y], Sin[x+y], f[], f[x, y, z]} /. f[t___] :> a*f[t]
{a f[x], g[x], a f[x, y], Sin[x+y], a f[], a f[x, y, z]}
```

Now the results of the rule substitutions for this input are the same.

- 4.2.4.15 A bit more useful example

To conclude this section, let us consider a bit more useful example of rule application : we will consider some polynomial of a single variable < x >, break it into a list of terms, and then replace all even powers of < x > by some object, say a literal < a > . Consider for example :

```
Clear[testexpr, x, a];
testexpr = Expand[(1+x)^10]
```

$1 + 10 x + 45 x^2 + 120 x^3 + 210 x^4 + 252 x^5 + 210 x^6 + 120 x^7 + 45 x^8 + 10 x^9 + x^{10}$

It is very easy to get a list of individual terms - let us look at the FullForm of this expression :

```
FullForm[testexpr]
```

```
Plus[1, Times[10, x], Times[45, Power[x, 2]], Times[120, Power[x, 3]],
 Times[210, Power[x, 4]], Times[252, Power[x, 5]], Times[210, Power[x, 6]],
 Times[120, Power[x, 7]], Times[45, Power[x, 8]], Times[10, Power[x, 9]], Power[x, 10]]
```

All that is needed in this particular case (but keep in mind that this is not a general solution, since it exploits the absence of any sums inside the terms) is to Replace the head Plus by head List :

```
testexpr /. Plus → List
```

$\{1, 10\,x, 45\,x^2, 120\,x^3, 210\,x^4, 252\,x^5, 210\,x^6, 120\,x^7, 45\,x^8, 10\,x^9, x^{10}\}$

Now we replace all even powers of < x > by < a > . Here we will go a little ahead of time and use a restricted pattern x^_?EvenQ. For now, let me just say that the pattern x^_ would mean any power of x (except perhaps just x, as it is represented differently), and x^_?EvenQ means any even power.

```
testexpr /. Plus → List /. x^(_?EvenQ) -> a
```

$\{1, 10\,x, 45\,a, 120\,x^3, 210\,a, 252\,x^5, 210\,a, 120\,x^7, 45\,a, 10\,x^9, a\}$

We can also, for instance, apply some function < f > to these even powers :

```
Clear[f, x, y];
testexpr /. x^(y_?EvenQ) :> f[x^y]
```

$1 + 10\,x + 120\,x^3 + 252\,x^5 + 120\,x^7 + 10\,x^9$
$+ 45\,f[x^2] + 210\,f[x^4] + 210\,f[x^6] + 45\,f[x^8] + f[x^{10}]$

The zero - th power (which is, the x - independent term) was missed. To account for it is not completely trivial. Here is the solution for this particular case - it contains too many details not covered yet so I will postpone the explanation.

```
Clear[f, x, y, a, b, newexpr];
newexpr = Replace[testexpr,
  {(a : _ : 1) * x^(y_?EvenQ) ⧴ a * f[x^y], a_ /; FreeQ[a, x] ⧴ f[a]}, 1]
```

$10\,x + 120\,x^3 + 252\,x^5 + 120\,x^7 + 10\,x^9 + f[1]$
$+ 45\,f[x^2] + 210\,f[x^4] + 210\,f[x^6] + 45\,f[x^8] + f[x^{10}]$

The function < f > can of course be anything. For instance, we may consider a shift by a constant < b > :

```
newexpr /. {f[z_^k_] ⧴ (z - b)^k, f[1] → 1}
```

$1 + 10\,x + 120\,x^3 + 252\,x^5 + 120\,x^7 + 10\,x^9 + 45\,(-b + x)^2$
$+ 210\,(-b + x)^4 + 210\,(-b + x)^6 + 45\,(-b + x)^8 + (-b + x)^{10}$

```
Clear[testexpr, newexpr];
```

▪ 4.2.5 Applying rules repeatedly - the ReplaceRepeated function

Since any given rule or a list of rules are normally tried on any element of expression just once, they don’t "keep track" of changes in an expression caused by their own actions. One may create more interesting constructs by repeatedly applying a rule or list of rules to an expression until it stops changing. By doing so, one in fact can imitate locally (and in a very oversimplified manner) the global evaluation process that *Mathematica* goes through in evaluating expressions. There exists a special built-in function performing such repeated rule application - **ReplaceRepeated**. Its symbolic equivalent is **//.** (slash-slash-dot). Let us give some examples:

▪ 4.2.5.1 Example: sorting a list of numbers

Let us generate some list of random integer numbers :

```
Clear[testlist];
testlist = Table[Random[Integer, {1, 20}], {15}]
```

```
{9, 1, 15, 18, 6, 18, 4, 10, 10, 11, 9, 8, 19, 18, 13}
```

This is the rule we need :

```
sortrule = {x___, y_, z_, t___} /; y > z :→ {x, z, y, t}
```

```
{x___, y_, z_, t___} /; y > z :→ {x, z, y, t}
```

What it does is clear from its form: it exchanges adjacent elements if the one to the right is smaller. Let us apply it:

```
testlist /. sortrule
```

```
{1, 9, 15, 18, 6, 18, 4, 10, 10, 11, 9, 8, 19, 18, 13}
```

```
testlist /. sortrule /. sortrule
```

```
{1, 9, 15, 6, 18, 18, 4, 10, 10, 11, 9, 8, 19, 18, 13}
```

It is clear that this is a case for ReplaceRepeated :

```
testlist //. sortrule
```

```
{1, 4, 6, 8, 9, 9, 10, 10, 11, 13, 15, 18, 18, 18, 19}
```

We have just obtained a rule-based realization of the exchange sort.

▪ 4.2.5.2 Example: deleting duplicate elements

Let us again generate a list:

```
Clear[testlist];
testlist = Table[Random[Integer, {1, 5}], {15}]

{5, 4, 5, 5, 5, 4, 1, 5, 3, 2, 2, 4, 3, 4, 3}
```

Suppose that we have to delete duplicate elements. This is the rule we need:

```
delrule = {x___, y_, z___, y_, t___} :→ {x, y, z, t}

{x___, y_, z___, y_, t___} :→ {x, y, z, t}
```

Let us run it a few times:

```
testlist /. delrule

{5, 4, 5, 5, 4, 1, 5, 3, 2, 2, 4, 3, 4, 3}
```

```
testlist /. delrule /. delrule
{5, 4, 5, 4, 1, 5, 3, 2, 2, 4, 3, 4, 3}
```

We see that it works so far. We now use ReplaceRepeated

```
testlist //. delrule
{5, 4, 1, 3, 2}
```

4.2.5.3 Example - a rule-based factorial

Here we will have the following rules:

```
Clear[fact];
frules = {fact[1] → 1, fact[n_Integer] :→ n * fact[n - 1]};
```

Let us check:

```
fact[5] /. frules
5 fact[4]
```

```
fact[5] /. frules /. frules

20 fact[3]
```

Finally,

```
fact[5] //. frules
120
```

Note that the rules are local. In particular, the expression fact[5] by itself does not have any value neither before, not after the application of rules:

```
fact[5]
fact[5]
```

Note also that had we placed the rules in a different order, and we would have entered an infinite loop, since the first rule would always apply and thus the second (fact[1]->1) would have no chance to apply. You can try this if you wish, but be sure to save your session and be ready to rerun the kernel.

```
Clear[frules];
```

■ 4.2.5.4 Efficiency issues

There are many other non-trivial examples of this technique. However, often it turns out to be not the most efficient one, since it is quite easy to build inefficient rules and patterns. For instance, our first example with list sorting has a terrible performance and is completely impractical for any realistic sizes of a list, since the pattern-matcher needs roughly linear (in the list size) time to find a first match for exchange, and then it only does a single exchange and starts all over! The number of exchanges needed is of the order of the square of the list size, and thus we conclude that our realization has roughly cubic complexity.

Let us do some performance measurements:

```
Clear[testlist];
testlist = Table[Random[Integer, {1, 500}], {25}];
```

This is the rule we need :

```
testlist //. sortrule; // Timing
{0.01, Null}
testlist = Table[Random[Integer, {1, 500}], {50}];
testlist //. sortrule; // Timing
{0.11, Null}
testlist = Table[Random[Integer, {1, 500}], {100}];
testlist //. sortrule; // Timing
{0.841, Null}
```

These timing results confirm our expectations. While this shows that our rule - based realization is completely inefficient since it adds another power of the list size to the standard complexity of the exchange sort algorithm, it is a good news that we can understand why this happens. Because it turns out that in many cases, the structures on which patterns are tried plus patterns themselves can be organized in such a way that the pattern is usually matched very soon after the beginning. In fact, as was demonstrated for instance by David Wagner in his book [7] in the context of the mergesort algorithm, this technique allows to make the rule-based solution the most efficient of all.

So, to put it simple: organize your data such as to ensure that the pattern matcher wastes as little time on a-priory doomed matching attempts as possible, and you will get an efficient rule-based solution.

■ 4.2.6 Conditional (restricted) patterns

All simple patterns are completely syntax - based. In many cases, we would like to make a decision - whether or not for the pattern to match - not just on the basis of its syntax but also checking certain conditions that the matched (sub) expressions must satisfy. This is when restricted or conditional patterns come handy.

Conditional patterns are just normal patterns, but with some condition attached to them. There are three main forms of conditional patterns - patterns of the form <**x_f**> which check the head of expression (we have already encountered those), patterns of the form **x_?Predicate** and patterns of the form **x/;condition**. We will now consider each type in more detail.

■ 4.2.6.1 Patterns which check the head of an expression

Since I already described these, let us just consider a few more examples.

- *Example 1*

Here is a list:

```
Clear[testlist, a, x];
testlist = {Pi, 1.5, 3 / 2, 10, y^2, ABC, Sin[x], 15, Cos[Exp[Tan[2]]]}
```

$\{\pi, 1.5, \frac{3}{2}, 10, y^2, \text{ABC}, \text{Sin}[x], 15, \text{Cos}[e^{\text{Tan}[2]}]\}$

We will now replace all integer numbers here by some object < a > (this can be anything) :

```
testlist /. _Integer :→ a
```

$\{\pi, 1.5, \frac{3}{2}, a, y^a, \text{ABC}, \text{Sin}[x], a, \text{Cos}[e^{\text{Tan}[a]}]\}$

Or, we can apply some function to these objects, but then we will need a tag for the pattern:

```
testlist /. x_Integer :→ f[x]
```

$\{\pi, 1.5, \frac{3}{2}, f[10], y^{f[2]}, \text{ABC}, \text{Sin}[x], f[15], \text{Cos}[e^{\text{Tan}[f[2]]}]\}$

- *Example 2*

We want to create a rule which will reverse the string. There is a built-in StringReverse function, so our first attempt is :

```
{x, 1, ABC, "never", Pi} /. s_ :→ StringReverse[s]
```

```
StringReverse::string : String expected at position 1 in StringReverse[x].

StringReverse::string : String expected at position 1 in StringReverse[1].

StringReverse::string : String expected at position 1 in StringReverse[ABC].

General::stop : Further output of StringReverse::string will be suppressed during this calculation. ≫
```

```
{StringReverse[x], StringReverse[1],
 StringReverse[ABC], reven, StringReverse[π]}
```

We see that we have to restrict the pattern to work only on real strings. We recall that all strings are atoms and have a head String (see section 1.1.5). Thus, we write :

```
{x, 1, ABC, "never", Pi} /. s_String :→ StringReverse[s]
{x, 1, ABC, reven, π}
```

- *Example 3*

We now want to apply a Sine function to all expressions which are of the form <f[something]>, where f is a fixed symbol. For example, for this list of expressions:

```
Clear[testlist, f, g];
testlist = {g[x], x^2, f[x], Cos[x * Tan[y]],
  f[x, y, z], f[z * Tan[x + y]], h[x, y, z], f[ArcSin[x - y]]}
```

$\{g[x], x^2, f[x], Cos[x\,Tan[y]], f[x, y, z],$
$f[z\,Tan[x+y]], h[x, y, z], f[ArcSin[x-y]]\}$

This may be done in many ways, but the simplest is just this :

```
testlist /. x_f → Sin[x]
```

$\{g[x], x^2, Sin[f[x]], Cos[x\,Tan[y]], Sin[f[x, y, z]],$
$Sin[f[z\,Tan[x+y]]], h[x, y, z], Sin[f[ArcSin[x-y]]]\}$

- *Example 4*

The last example on this topic: say we have a function <f>, which has to be defined so that any of its powers is equal to itself: f[f[f[f[...f[x]]]]] = f[x]. This is the rule to do it:

```
f[x_f] :→ x
```

This rule though has to be used with ReplaceRepeated (//.) rather than with ReplaceAll :

```
testlist = NestList[f, x, 5]
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]], f[f[f[f[f[x]]]]]}
testlist /. f[x_f] :→ x
{x, f[x], f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

```
testlist /. f[x_f] :→ x /. f[x_f] :→ x
{x, f[x], f[x], f[x], f[f[x]], f[f[f[x]]]}
testlist //. f[x_f] :→ x
{x, f[x], f[x], f[x], f[x], f[x]}
```

Such properties are characteristic to say projectors, and by defining rules like this, we may eliminate a lot of unnecessary work, in the case when these symbolic transformations are carried out before any specific representation of a projector (say, a matrix or a kernel of an integral operator) is used.

The checks of this type (head checks) are most frequently used to implement type - checks in function definitions, since they allow us to trivially narrow down the sets of objects on which this or that function has to be defined (if these sets can be identified with a certain head). For example, a pattern <x_List> will match only lists, <x_String> - only strings, etc. Moreover, one can define a new data type by considering a "container" of the form < newdata[data definitions] > . Then, it is trivial to arrange that the functions defined on this data type will only work on the proper object - one just has to use patterns like <x_newdata>. And since this check is entirely syntactic, there is no performance overhead induced by this procedure.

■ 4.2.6.2 Patterns which check some condition - commands Condition and PatternTest

This is a more general type of patterns. If the simple pattern were <x_>, then the conditional pattern may look like <x_?ftest> or <x_/;ftest[x]>. In both cases, <ftest> stands for a name of a predicate function which checks certain condition and returns True or False. In the first case, the question mark is a short hand notation for the built-in command **PatternTest**:

```
PatternTest[x_, ftest]
x_ ? ftest
```

In the second case, the combination < **/;** > (slash - semicolon) is a short - hand notation for the built - in function **Condition**:

```
Condition[x_, ftest[x]]
x_ /; ftest[x]
```

We will see that the PatternTest is less general than Condition. In particular, when the pattern contains more than one pattern tag, PatternTest usually can not be used, but Condition can, and we can impose conditions which depend on several pattern tags.

One important point about conditional pattern - matching is that the "presumption of guiltiness" is in effect here : if the condition (implemented either through Condition or PatternTest) evaluates to neither True nor False, the pattern does not match (as if we were using TrueQ). This requires extra care in pattern construction, especially in cases such as when the pattern is used to match elements which have to be deleted from a list if pattern does not match. This property can also sometimes be used to one's advantage. One such non-trivial application of this behavior is found in implementing function's error messages when writing packages.

Now consider some examples.

- *Example 1*

Here we will take a rather arbitrary list of numbers and raise all integer numbers in it to the cubic power.

```
Clear[testlist];
testlist = Table[Random[Integer, {-10, 10}] / 2, {15}]
```

$$\left\{0, -3, \frac{3}{2}, -4, -5, 3, \frac{5}{2}, -3, 0, \frac{1}{2}, -4, \frac{3}{2}, 4, \frac{1}{2}, 4\right\}$$

This is the solution:

```
testlist /. x_Integer ? Positive :> x^3
```

$$\left\{0, -3, \frac{3}{2}, -4, -5, 27, \frac{5}{2}, -3, 0, \frac{1}{2}, -4, \frac{3}{2}, 64, \frac{1}{2}, 64\right\}$$

Or :

```
testlist /. x_Integer /; Positive[x] :> x^3
```

$$\left\{0, -3, \frac{3}{2}, -4, -5, 27, \frac{5}{2}, -3, 0, \frac{1}{2}, -4, \frac{3}{2}, 64, \frac{1}{2}, 64\right\}$$

Or :

```
testlist /. x_ /; (IntegerQ[x] && Positive[x]) :> x^3
```

$$\left\{0, -3, \frac{3}{2}, -4, -5, 27, \frac{5}{2}, -3, 0, \frac{1}{2}, -4, \frac{3}{2}, 64, \frac{1}{2}, 64\right\}$$

```
Clear[testlist];
```

- *Example 2*

Here we will choose from the list of integers only those numbers which are equal to 2 modulo 5, or, rather, eliminate all other numbers from the list. Here we will go a little ahead of time and define our own predicate function :

```
Clear[ourtest];
ourtest[x_Integer] := Mod[x, 5] =!= 2;
```

Now we generate a list :

```
Clear[testlist, a, x];
testlist = Table[Random[Integer, {1, 20}], {15}]
```

```
{7, 15, 9, 12, 3, 16, 18, 2, 16, 17, 13, 5, 5, 20, 4}
```

With the tools we have now, we can do it with ReplaceRepeated and the following pattern :

```
testlist //. {x___, y_Integer /; ourtest[y], z___} :> {x, z}
```

```
{7, 12, 2, 17}
```

However, this is terribly inefficient, since to delete a single element, the whole run of the pattern - matcher through the list is required (in general, the match will happen somewhere in the middle of the list). There is a better rule - based solution :

```
testlist /. x_ /; ourtest[x] :→ Sequence[]
{7, 12, 2, 17}
```

Or, which is the same, but even more concise

```
testlist /. x_ ? ourtest :→ Sequence[]
{7, 12, 2, 17}
```

What happens here is that every element in a list is tried in a single run of the pattern-matcher, and if it has to be eliminated, it is replaced by Sequence[]. The reason that we don't see Sequence[] objects in our resulting list is that the Sequence[] means "emptiness", or "absence of any elements", and it usually disappears inside any other head.

We can package this into a function :

```
Clear[deleteIf];
deleteIf[lst_List, test_] := lst /. x_ ? test :→ Sequence[];
```

Check :

```
deleteIf[testlist, ourtest]
{7, 12, 2, 17}
```

A few comments: first, the pattern tag < x > above is considered a local variable since RuleDelayed is used. Second, we have in fact written a *higher - order function*, that is, a function which takes another function as one of its arguments. Notice that in Mathematica this is completely effortless and does not require any particular syntax.

Efficiency analysis and a procedural version

The speed of the latter two realizations will be roughly the same(PatternTest may be slightly faster), but very different from the first version:

```
testlist = Table[Random[Integer, {1, 20}], {2000}];
testlist //. {x___, y_Integer /; ourtest[y], z___} :→ {x, z}; // Timing
{9.443, Null}
testlist /. x_ /; ourtest[x] :→ Sequence[]; // Timing
{0.02, Null}
testlist /. x_ ? ourtest :→ Sequence[]; // Timing
{0.01, Null}
```

This is by the way the analogous procedural code needed to get the same result.

```
Module[{i, j, newlist},
   For[i = j = 1; newlist = testlist,
    i ≤ Length[testlist], i ++, If[Not[ourtest[testlist[[i]]]],
     newlist[[j ++]] = testlist[[i]]]];
   newlist = Take[newlist, j -1];
   newlist]; // Timing
```

```
{0.07, Null}
```

Not only it is much clumsier and needs an introduction of auxiliary variables, but it is also a factor of 5 - 7 slower than our best rule - based solution. I give here this comparison to show that the rule - based programming is not necessarily slow, but may in fact be the fastest way to obtain a result.

So, the moral of this example is the following: in *Mathematica*, the rule-based approach has a potential to outperform procedural approach by a wide margin, and also to be much more concise and intuitive. But it also can be misused badly if inefficient patterns are used. The pattern is almost certainly inefficient if the pattern-matcher is able to match and transform only one or few items in a large collection (list) of them, in a single run through this list. For efficient programs, the patterns like {x___,a_,y___,b_,z___) in conjunction with ReplaceRepeated should be avoided.

In some sense, a statement like "rule-based programming is slow in *Mathematica*" does not make any sense since all programming in *Mathematica* is rule - based to some extent.

- *Example 3*

Suppose we now want to perform some actions (say, take a square root) only on those elements in a list of integers, which are the full squares. First thing that comes to mind is to try a pattern x_^2:

```
Clear[testlist];
testlist = Range[30]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
```

```
testlist /. x_^2 -> Sqrt[x]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
```

Nothing happened. The reason is that even if a number is a full square, its internal form is not a^2. Since the pattern - matching for simple patterns is based on syntax only, no matches occurred. This is what we have to use instead :

```
testlist /. x_ /; IntegerQ[Sqrt[x]] → {Sqrt[x]}
```

```
{{1}, 2, 3, {2}, 5, 6, 7, 8, {3}, 10, 11, 12, 13, 14, 15,
 {4}, 17, 18, 19, 20, 21, 22, 23, 24, {5}, 26, 27, 28, 29, 30}
```

I placed the transformed numbers into extra list braces to make them visible.

- *Example 4*

There is a built-in predicate PrimeQ in *Mathematica*, which checks if a number is a prime. We can use it to collect say all the primes in the first 1000 natural numbers:

```
Range[1000] /. x_Integer /; Not[PrimeQ[x]] ⧴ Sequence[] // Short
{2, 3, 5, 7, ≪161≫, 983, 991, 997}
```

This is not the absolutely fastest way to do it (which perhaps will be using the Select function if we are willing to sweep through every number and use PrimeQ), but not the worst either.

- *Example 5*

One can impose more complicated conditions, and on more complicated patterns. Let us now create a list of lists of 2 random integers:

```
Clear[testlist];
testlist = Partition[Table[Random[Integer, {1, 20}], {20}], 2]
{{4, 20}, {5, 11}, {19, 11}, {14, 20},
 {9, 19}, {7, 3}, {14, 19}, {20, 19}, {9, 12}, {9, 4}}
```

Note that we used the Partition command. We could instead use Table with 2 iterators. Now, we would like to exchange the numbers in any small sublist where the first number is even and the second is odd. Here is the rule we need:

```
exchangerule = {x_, y_} /; (EvenQ[x] && OddQ[y]) ⧴ {y, x};
```

Check :

```
testlist
testlist /. exchangerule


{{4, 20}, {5, 11}, {19, 11}, {14, 20},
 {9, 19}, {7, 3}, {14, 19}, {20, 19}, {9, 12}, {9, 4}}
{{4, 20}, {5, 11}, {19, 11}, {14, 20},
 {9, 19}, {7, 3}, {19, 14}, {19, 20}, {9, 12}, {9, 4}}
Clear[testlist, delrule, exchangerule];
```

■ 4.2.7 Alternative patterns

Often the same type of transformations have to be carried out with different expressions. This means, that for expressions of different types, we would normally need several rules which are essentially different only in their left hand side.

As an example, say we need to apply some rule to either integer or rational numbers, and let this transformation be to take a square root of these numbers. For the following test list of expressions:

```
{x, 2, Pi, 3 / 2, 2 / 5, 4, Sin[y], 8, Cos[z]}
```

We could use conditional patterns to do this :

```
{x, 2, Pi, 3 / 2, 2 / 5, 4, Sin[y], 8, Cos[z]} /.
 x_ /; Head[x] === Integer || Head[x] === Rational :→ Sqrt[x]
```

$\left\{x, \sqrt{2}, \pi, \sqrt{\frac{3}{2}}, \sqrt{\frac{2}{5}}, 2, \text{Sin}[y], 2\sqrt{2}, \text{Cos}[z]\right\}$

However, this solution is not the most elegant and concise, and more importantly, not the most efficient. *Mathematica* provides a mechanism to group different patterns together and form so called alternative patterns. The built - in command which does this has a literal equivalent Alternatives and a short - hand notation | :

```
Alternatives[a, b]
```

a | b

With it, we can solve our problem like this :

```
{x, 2, Pi, 3 / 2, 2 / 5, 4, Sin[y], 8, Cos[z]} /.
 x_Integer | x_Rational :→ Sqrt[x]
```

$\left\{x, \sqrt{2}, \pi, \sqrt{\frac{3}{2}}, \sqrt{\frac{2}{5}}, 2, \text{Sin}[y], 2\sqrt{2}, \text{Cos}[z]\right\}$

Alternative patterns are quite often used. Some especially sleek applications of them usually employ a mixture of rule-based and functional programming. We will see many examples of them in later chapters.

■ 4.2.8. Giving names to entire patterns - the Pattern command

It is possible and often quite useful to give names to entire patterns. To do this, one should just use the built - in **Pattern** command, which has a short - hand notation < **:**> (colon). This is how the previous problem can be solved by using Pattern:

```
{x, 2, Pi, 3 / 2, 2 / 5, 4, Sin[y], 8, Cos[z]} /.
 x : (_Integer | _Rational) :→ Sqrt[x]
```

$\left\{x, \sqrt{2}, \pi, \sqrt{\frac{3}{2}}, \sqrt{\frac{2}{5}}, 2, \text{Sin}[y], 2\sqrt{2}, \text{Cos}[z]\right\}$

We will frequently use this form in later chapters.

■ 4.2.9 Optional patterns

The default pattern is a construction which allows to modify a given pattern to match not just what it normally matches, but also certain expressions in which some parts of the pattern will be missing altogether, but then the pattern "knows" what to substitute for them. This is often useful for defining functions with some special cases (default values for some input parameters).

The default pattern is built with the **Optional** keyword, and with a colon as a short-hand notation. The way it is used is <pattern:defvalue>, where the <defvalue> is substituted if this particular piece of the pattern is absent in an expression, which otherwise matches the pattern.

Here, for example, we want to replace all sequences of arguments in < f > by their sum, and for a single argument, add 1.

```
f[{a}, {a, b}, {a, b, c}]
```

We can do this by two separate rules :

```
f[{a}, {a, b}, {a, b, c}] /. {{x_} :> {x + 1}, {x__} :> {Plus[x]}}
f[{1 + a}, {a + b}, {a + b + c}]
```

However, first, we needed 2 rules, and second, we would get a bug if we did not think of the right order of the rules and interchange them, since the more general pattern also matches on a single argument :

```
f[{a}, {a, b}, {a, b, c}] /. {{x__} :> {Plus[x]}, {x_} :> {x + 1}}
f[{a}, {a + b}, {a + b + c}]
```

Here is the solution with an optional pattern :

```
f[{a}, {a, b}, {a, b, c}] /. {x__, y_ : 1} :> {x + y}
f[{1 + a}, {a + b}, {a + b + c}]
```

■ 4.2.10 Repeated patterns

These patterns are useful in cases when we have to match a sequence of expressions each of which is matched by a given pattern, but we don't know how many terms there will be in a sequence.

Here we make a repeated pattern which matches any sequence of rational or integer numbers, mixed in any way

```
f[3 / 5, 4, 5, 2 / 3] /. f[x : (_Integer | _Rational) ..] :> {x}
```

$\left\{\frac{3}{5}, 4, 5, \frac{2}{3}\right\}$

Here we convert to numbers all lists which represent binary digits - that is, all lists which contain any number of ones and zeros mixed arbitrarily.

```
{{1, 0, 0}, {0, 1, 0}, {1, 1, 1}, {2, 0, 1}, {1, 0}, {1, 0, 3}} /.
  x : {(1 | 0) ..} :> FromDigits[x, 2]
{4, 2, 7, {2, 0, 1}, 2, {1, 0, 3}}
```

4.3 Built-in functions that use patterns

Let us look at some highly useful built-in functions which take patterns as some of their arguments. One such function - MatchQ - we already discussed.

4.3.1 Cases

This function is used for expression destructuring. More precisely, it is used to search for subexpressions inside an expression, which match certain pattern. **Cases[expr, pattern]** returns all elements of <expr> on the level 1, which match the pattern <pattern>. As an optional third argument Cases takes a level specification, which can be an integer (positive or negative, including Infinity), an integer in list braces, or a pair of integers in a list. In the first case, Cases performs search on all levels up to and including the indicated number (from the top or from the bottom of the expression tree, depending on its sign), in the second case it only searches on the indicated level only, and in the third case it searches in the range of levels given by the two numbers in the list. As an optional fourth argument, it accepts an integer indicating how many results have to be found until Cases stops. If it is not given, Cases will produce all the results on given level(s) of expression.

Now some examples :

4.3.1.1 Example: choosing integers from the list

The simplest example: let us choose all integer numbers from a simple (1-dimensional) list:

```
Clear[testlist];
testlist = {3 / 2, Pi, 3, 1.4142135, 10, 99.99, 15, 25};

Cases[testlist, _Integer]
```

```
{3, 10, 15, 25}
```

Notice that here we don't even need to attach a tag to the pattern, since we don't need to perform any transformations on the results.

4.3.1.2 Example: filtering data

Let us consider a bit more practical example : suppose we have a set of data and need to remove all the data points with values smaller than a given cutoff < eps > "

```
eps = 0.3;
data = Table[Random[], {10}]
```

```
{0.558611, 0.087393, 0.699237, 0.393591, 0.815213,
 0.251073, 0.756381, 0.383772, 0.24806, 0.721713}
```

This is how one does this with Cases :

```
Cases[data, x_ /; x > eps]
{0.558611, 0.699237, 0.393591,
 0.815213, 0.756381, 0.383772, 0.721713}
```

■ 4.3.1.3 Extended functionality

One more capability that Cases has is to immediately perform some transformation on the result found. The syntax is **Cases[expr, rule, levespec, n]**, with the last two arguments optional as before, and we see that where we had a pattern now is a rule, with a pattern being its l.h.s. For example, now we want to supply each found number with either True or False depending on whether or not a number is divisible by 5:

```
Cases[testlist, x_Integer :> {x, Mod[x, 5] == 0}]
{{3, False}, {10, True}, {15, True}, {25, True}}
```

So, notice that Cases generates a list as a result of its execution. This kind of list generation we may call a dynamic list generation rather than the "static" one we have described in the previous chapter.

```
Clear[testlist];
```

■ 4.3.1.4 Example : selecting positive numbers in a list

Let us now generate a list of positive and negative random integers:

```
Clear[testlist];
testlist = Table[Random[Integer, {-10, 10}], {15}]
{-9, -4, 2, 4, -2, -7, 10, -6, -4, 6, -9, 3, -4, -4, 5}
```

Let us select only positive ones - the pattern we need is _?Positive

```
Cases[testlist, _?Positive]


{2, 4, 10, 6, 3, 5}
```

Now only those that are larger than 5:

```
Cases[testlist, x_ /; x > 5]


{10, 6}
```

Or those which are divisible by 3:

```
Cases[testlist, x_ /; Mod[x, 3] == 0]
{-9, -6, 6, -9, 3}
```

In the last 2 cases we have used the conditional patterns.

■ 4.3.1.5 Example: nested list

Let us now consider a more complex 2-level list:

```
Clear[testlist];
testlist = Table[i + j, {i, 1, 4}, {j, 1, 3}]

{{2, 3, 4}, {3, 4, 5}, {4, 5, 6}, {5, 6, 7}}
```

Now say we would like to find all even numbers. Let us try:

```
Cases[testlist, x_ ? EvenQ]
{}
```

It did not work ... This is because by default, Cases only looks at the first level of an expression, where there are no numbers, just sublists.

```
FullForm[testlist]

List[List[2, 3, 4], List[3, 4, 5], List[4, 5, 6], List[5, 6, 7]]
```

This is how we should search :

```
Cases[testlist, x_ ? EvenQ, 2]

{2, 4, 4, 4, 6, 6}
```

The argument < 2 > here instructs Cases to search on all levels up to level 2 inclusive, on the first and second levels in this case. Should we wish to search only on a level 2, we would have to place it in a curly braces (list) :

```
Cases[testlist, x_ ? EvenQ, {2}]
{2, 4, 4, 4, 6, 6}
```

In this case it did not matter, but say now we want to find not numbers but sublists. We could do it like this :

```
Cases[testlist, _List]

{{2, 3, 4}, {3, 4, 5}, {4, 5, 6}, {5, 6, 7}}
```

This looks just like the initial list, but really it is a list of found sublists, which in this case indeed coincides with the initial list. This is easy to check by transforming the results:

```
Clear[found];
Cases[testlist, x_List :> found[x]]

{found[{2, 3, 4}], found[{3, 4, 5}],
 found[{4, 5, 6}], found[{5, 6, 7}]}
```

We can impose some conditions on the sublists - for example, to find only those which contain number 4 :

```
Cases[testlist, x_List /; Cases[x, 4] =!= {}]
```

```
{{2, 3, 4}, {3, 4, 5}, {4, 5, 6}}
```

This illustrates a possibility to use nested Cases like this (however, there are more efficient solutions for the present case, such as using the MemberQ function - see below).

If we now restrict ourselves to search only on the level 2,

```
Cases[testlist, _List, {2}]
```

```
{}
```

then we find nothing, since there are no (sub)lists on level 2, just numbers.

- 4.3.1.6   Example: an elegant solution of the problem of odd sublists from chapter 3

As an example of combined use of Cases and conditional patterns, we can revisit a problem of extracting from a list all sublists which contain odd number of odd elements. Before we considered procedural, structural and functional implementations (see section 3.6.8.4). Our present solution will be completely pattern-based.

Here is our list:

```
Clear[testlist];
testlist = Range /@ Range[6]
```

```
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

And here is the solution:

```
Cases[testlist, x_List /; OddQ[Count[x, _ ? OddQ]]]
```

```
{{1}, {1, 2}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
```

In my opinion, this is the more elegant way to solve this problem. Also, it is very transparent what is being done: in each list we count the number of odd elements with a command Count[x,_?OddQ] (the Count command we will cover shortly), then we check whether or not the resulting number is odd. I remind that /; symbol is a short - hand for Condition operator.

As a modification, we may do the same but for each found sublist return it together with the number of even elements in it:

```
Cases[testlist,
 x_List /; OddQ[Count[x, _ ? OddQ]] :> {Count[x, _ ? EvenQ], x}]
```

```
{{0, {1}}, {1, {1, 2}}, {2, {1, 2, 3, 4, 5}}, {3, {1, 2, 3, 4, 5, 6}}}
```

```
Clear[testlist];
```

■ 4.3.1.7 Example: first n prime numbers in a list

Say we want to get a first given number of primes in a list of numbers. This is our list:

```
testlist = Table[Random[Integer, {1, 100}], {30}]
{94, 27, 84, 57, 40, 12, 20, 76, 35, 47, 51, 62, 14, 71, 16,
 87, 20, 29, 52, 46, 56, 20, 42, 18, 94, 71, 73, 86, 54, 99}
```

This will pick the first 3 primes :

```
Cases[testlist, _?PrimeQ, 1, 3]
{47, 71, 29}
```

Notice that here we still had to provide the third argument (level specification), even though it is normally unnecessary for simple lists (it is 1 by default). This is because otherwise the argument <3> would be interpreted as a level specification, and as a result, we would get all primes (why?)

```
Cases[testlist, _?PrimeQ, 3]
{47, 71, 29, 71, 73}
```

■ 4.3.1.8 Why not use loops?

One may think that in all the examples given above using loops will give the same effect. But this is not so at least for 3 reasons:

1. Cases is optimized in terms of generation of lists of results - this list is generated internally. As we have seen in section 3.4.5.3, generating a list in a loop is quite inefficient.
2. Cases works on patterns, and selects elements based on pattern-matching rather than a simple comparison. Of course, in the case when we simply look for a fixed object (not containing a pattern), pattern-matching reduced to the sameness comparison.
3. Cases works on general *Mathematica* expressions (trees), and we can specify on which levels of the trees the search has to be performed. This would require nested loops in the procedural approach.

■ 4.3.1.9 Example: collecting terms in a polynomial of 2 variables

We could have given many more examples. What is important to remember is that Cases is a universal and versatile command, which works on general *Mathematica* expressions (trees), and on lists in particular. To illustrate the generality of Cases, let us now use it to select from the polynomial (1+x)^10 *(1+y)^10 all terms with an even power of <y> and odd power of <x>:

```
Clear[a, x, y];
Cases[Expand[(1 + x)^10 * (1 + y)^10],
  a_ * x^_?OddQ * y^_?EvenQ, Infinity]
```

$\{5400\,x^3\,y^2,\ 11340\,x^5\,y^2,\ 5400\,x^7\,y^2,\ 450\,x^9\,y^2,\ 25200\,x^3\,y^4,\ 52920\,x^5\,y^4,\ 25200\,x^7\,y^4,\ 2100\,x^9\,y^4,\ 25200\,x^3\,y^6,\ 52920\,x^5\,y^6,\ 25200\,x^7\,y^6,\ 2100\,x^9\,y^6,\ 5400\,x^3\,y^8,\ 11340\,x^5\,y^8,\ 5400\,x^7\,y^8,\ 450\,x^9\,y^8,\ 120\,x^3\,y^{10},\ 252\,x^5\,y^{10},\ 120\,x^7\,y^{10},\ 10\,x^9\,y^{10}\}$

The last argument set to Infinity means that all levels of an expression should be searched.

4.3.2 DeleteCases

As is it clear from the name of this function, it deletes from a list all the elements which match a given pattern. Its syntax is similar to that of Cases. The main and important difference is that Cases returns a list of found subexpressions, while DeleteCases returns the (copy of) original list with these subexpressions removed.

4.3.2.1 Example: deleting odd numbers from a list

Here we will delete all odd numbers from a list.

```
testlist = Range[15];
DeleteCases[Range[15], _ ? OddQ]


{2, 4, 6, 8, 10, 12, 14}
```

We have just removed all odd numbers from the list. But this does not mean that <testlist> itself changed in any way:

```
testlist
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

In this respect, DeleteCases works the same as the majority of *Mathematica* built - in commands, that is - without side-effects : the copy of the input variable is created and modified. Should we wish to change the content of <testlist>, we have to write:

```
testlist = DeleteCases[testlist, _ ? OddQ]
{2, 4, 6, 8, 10, 12, 14}
```

Let us perform a small timing measurement : measure the time it will take DeleteCases to remove from the list of first 100000 natural numbers those whose remainder of division by 5 is smaller than 2 :

```
Timing[Short[DeleteCases[Range[100 000], x_ /; Mod[x, 5] ≤ 2]]]
{0.531, {3, 4, <<39 996>>, 99 998, 99 999}}
```

Here is a straightforward procedural realization :

```
Module[{i, j, result, starting},
   For[i = j = 1; starting = result = Range[100 000],
    i ≤ 100 000, i ++, If[Mod[starting[[i]], 5] ≤ 2,
     result[[j ++]] = starting[[i]]]];
   Take[result, j -1]] // Short // Timing
{2.063, {1, 2, <<59 996>>, 99 997, 100 000}}
```

It requires extra auxiliary variables that I had to localize with Module (will cover that later), is 5 times longer and about as many times slower. This confirms once again the main rule of working with lists in *Mathematica* - avoid breaking them into pieces.

4.3.2.2 Example: non-zero integer subsequences

Consider a following problem : we are given a list of integers, some of which can be zero. The task is to extract from the list all subsequences of consecutive non - zero elements.

For example, this will be our test data set :

```
testdata = Table[Random[Integer, {0, 3}], {20}]
{3, 0, 3, 1, 3, 1, 0, 0, 3, 2, 1, 2, 1, 2, 1, 3, 1, 3, 2, 3}
```

The first step in solving this problem will be to use Split (see section 3.10.3) to split the elements into sublists whenever zero is encountered :

```
step1 = Split[testdata, #1 != 0 &]
{{3, 0}, {3, 1, 3, 1, 0}, {0}, {3, 2, 1, 2, 1, 2, 1, 3, 1, 3, 2, 3}}
```

We have however captured some zeros into our sublists, so we have to delete them (note the level specification) :

```
step2 = DeleteCases[step1, 0, {2}]
{{3}, {3, 1, 3, 1}, {}, {3, 2, 1, 2, 1, 2, 1, 3, 1, 3, 2, 3}}
```

Finally, the previous operation in general produces some number of empty lists from sublists containing a single zero (there can not be sublists containing several zeros - why?). We have now to delete them as well:

```
step3 = DeleteCases[step2, {}]
{{3}, {3, 1, 3, 1}, {3, 2, 1, 2, 1, 2, 1, 3, 1, 3, 2, 3}}
```

We now package all the steps into a function :

```
Clear[nonzeroSubsequences];
nonzeroSubsequences[x : {__Integer}] :=
 DeleteCases[DeleteCases[Split[x, #1 != 0 &], 0, {2}], {}]
```

Note the use of a named pattern and BlankSequence, to better restrict the argument. Check :

```
nonzeroSubsequences[testdata]
{{3}, {3, 1, 3, 1}, {3, 2, 1, 2, 1, 2, 1, 3, 1, 3, 2, 3}}
```

■ 4.3.2.3 Cases and DeleteCases: similarities and differences

In terms of syntax, most operations with DeleteCases are very similar to those with Cases. Note however that they are usually used in logically very different situations. Cases is used when we need to extract some parts (subexpressions) from an expression, and we don’t really care "what will happen" to the expression itself afterwards. DeleteCases, in contrast, is used to perform structural changes of the expression itself (I remind that by expression itself I mean a copy of the input, created internally by DeleteCases - as usual in *Mathematica,* it does not introduce side effects and the original input is not modified in any way), when we don’t care what will happen with those pieces that we delete. In this sense, Cases and DeleteCases are exact opposites of each other.

```
Clear[testlist];
```

4.3.3 MemberQ

This function finds out whether an object (in general, pattern) is a part of some expression. In the case of pattern, it determines whether there are subexpressions matching this pattern. The format is

MemberQ[expr, pattern, levspec],

where the third optional argument < levspec > determines, as usual, the level (s) on which to perform the search. Examples :

- 4.3.3.1 Example: checking for presence of primes

```
MemberQ[{4, 6, 8, 9, 10}, _?PrimeQ]
False
```

Since there were no primes here, the result is False.

- 4.3.3.2 Example: testing membership in a symbolic list

```
Clear[a, b, c, d, e];
MemberQ[{a, b, c, d, e}, a]

True
```

- 4.3.3.3 Example: testing membership in nested lists

Consider now an example we already looked at, the one with nested lists:

```
Clear[testlist];
testlist = Range /@ Range[6]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5, 6}}
MemberQ[testlist, _Integer]
False
```

This happened because by default only the first level is considered. However,

```
MemberQ[testlist, _List]
True
```

since the elements of the first level are sublists. When we look at the second level :

```
{MemberQ[testlist, _Integer, {2}], MemberQ[testlist, _List, {2}]}

{True, False}
```

the result is the opposite, since the second level is populated by numbers. If we look on both levels though, then both checks will produce True :

```
{MemberQ[testlist, _Integer, 2], MemberQ[testlist, _List, 2]}
{True, True}
Clear[testlist];
```

■ 4.3.3.4 Example: Unsorted Intersection

As we know, there exists a built - in Intersection command which finds an intersection (common part) of two lists. However, it removes the duplicate elements and sorts the result, which is not always the desired behavior. We can use a combination of Cases and MemberQ to write our version which will not sort the result and will not delete identical entries. So, to put it simple : given the two lists, we have to keep in the first list only those elements that are present also in the second one.

Here are our lists :

```
list1 = Table[Random[Integer, {1, 30}], {20}]
list2 = Table[Random[Integer, {1, 30}], {20}]


{30, 13, 12, 15, 12, 1, 23, 26, 25, 26, 20, 5, 13, 30, 26, 12, 8, 6, 21, 6}
{20, 5, 23, 17, 7, 7, 2, 8, 15, 7, 19, 6, 28, 3, 15, 14, 12, 11, 28, 18}
```

And this is a solution :

```
Cases[list1, x_ /; MemberQ[list2, x]]
{12, 15, 12, 23, 20, 5, 12, 8, 6, 6}
```

Note that there exists and alternative version which does not involve MemberQ, but rather involves an alternative pattern:

```
Cases[list1, Apply[Alternatives, list2]]
{12, 15, 12, 23, 20, 5, 12, 8, 6, 6}
```

The meaning of the Apply operation will be explained in chapter V, but basically here it is used to create a large alternative pattern from the list :

```
Apply[Alternatives, list2]
20 | 5 | 23 | 17 | 7 | 7 | 2 | 8 | 15 | 7 | 19 | 6 | 28 | 3 | 15 | 14 | 12 | 11 | 28 | 18
```

It turns out that the second solution is generally more efficient for large lists. Note however, that both of them have in general the complexity L1*L2, where L1 and L2 are the lengths of lists. This fact may be somewhat hidden because commands like Cases and MemberQ are optimized, and the above solutions are certainly faster than a nested loop. They work well as "scripting" solutions for small lists, but for large lists there are more efficient implementations. We will discuss this at length in chapter VI.

```
Clear[list1, list2];
```

4.3.4 Position - a second look

We have already discussed this command in the previous chapter on lists. However, since at the time we did not cover patterns, our discussion there was rather limited. In fact, Position is a much more general operation since it works on arbitrary patterns, and returns all the positions in expression where the pattern matches. Let us consider a few examples.

4.3.4.1 Example: positions of numbers divisible by 3

```
Range[30]
Position[Range[30], x_ /; Mod[x, 3] == 0]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
{{3}, {6}, {9}, {12}, {15}, {18}, {21}, {24}, {27}, {30}}
```

4.3.4.2 An example with symbolic expression

```
expr = Expand[(1 + x)^10]
```

$1+10\,x+45\,x^2+120\,x^3+210\,x^4+252\,x^5+210\,x^6+120\,x^7+45\,x^8+10\,x^9+x^{10}$

```
FullForm[expr]
```

```
Plus[1, Times[10, x], Times[45, Power[x, 2]], Times[120, Power[x, 3]],
 Times[210, Power[x, 4]], Times[252, Power[x, 5]], Times[210, Power[x, 6]],
 Times[120, Power[x, 7]], Times[45, Power[x, 8]], Times[10, Power[x, 9]], Power[x, 10]]
```

```
Position[expr, x^_?OddQ]
```

```
{{4, 2}, {6, 2}, {8, 2}, {10, 2}}
```

Returned are the positions where odd powers of the variable <x> (excluding x itself) reside in an expression <expr>. We may use Extract to check this:

```
Extract[expr, Position[expr, x^_?OddQ]]
```

$\{x^3, x^5, x^7, x^9\}$

Equivalently, we could use Cases :

```
Cases[expr, x^_?OddQ, Infinity]
```

$\{x^3, x^5, x^7, x^9\}$

With positions, however, we can do more. For instance, if we remove the last index from each position in the position list, we extract expressions which include these powers as their parts :

```
Extract[expr,
 Position[expr, x^_ ? OddQ] /. {y__Integer, z_Integer} ⧴ {y}]
```

```
{120 x^3, 252 x^5, 120 x^7, 10 x^9}
```

Be sure to understand what we just did. By the way, this code presents an example of an (alternative to the pure pattern-based) mixed structural/pattern-based way to perform deconstruction of expressions.

The Position command will be also used in the more complicated example at the end of this section.

4.3.5. Count

The Count function Count[expr,pattern,levspec] counts occurrences of subexpressions in <expr> matching the pattern <pattern>, with a level specification given by an optional third argument <levspec>. Examples:

4.3.5.1 A simple example

This is a number of numbers divisible by 6, among the first 30 natural numbers

```
Count[Range[30], x_ /; Mod[x, 6] == 0]
```

```
5
```

4.3.5.2 Example: number of times a given letter is present in a sentence

A number of letters "s" in some phrase

```
chars = Characters["she sells sea shells on the sea shore"]
```

```
{s, h, e,  , s, e, l, l, s,  , s, e, a,  , s, h, e,
 l, l, s,  , o, n,  , t, h, e,  , s, e, a,  , s, h, o, r, e}
```

```
Count[chars, "s"]
```

```
8
```

Here are the unique characters in this phrase together with their frequencies:

```
alphabet = Union[chars];
freqchars = Table[{alphabet[[i]], Count[chars, alphabet[[i]]]},
  {i, Length[alphabet]}]
```

```
{{ , 7}, {a, 2}, {e, 7}, {h, 4},
 {l, 4}, {n, 1}, {o, 2}, {r, 1}, {s, 8}, {t, 1}}
```

We can sort this list in the order of increasing frequencies:

```
Sort[freqchars, #1[[2]] < #2[[2]] &]
```

```
{{t, 1}, {r, 1}, {n, 1}, {o, 2},
 {a, 2}, {l, 4}, {h, 4}, {e, 7}, { , 7}, {s, 8}}
```

Notice the use of the custom sorting functions, which compares second elements of the sublists (frequencies). The pure function used here will be covered soon.

```
Clear[chars, freqchars, alphabet];
```

Recall also that we used Count before, in a problem with odd sublist extraction (section 3.6.8.4).

4.3.6 FreeQ

This command is used to test whether a given expression is completely free of some symbol or subexpression - that is, does not contain it as a subexpression. The format is **FreeQ[expr,pattern]**. As its name suggests, FreeQ is a predicate returning True or False. It is quite useful in cases when new rules for some object have to be defined - a classic example being the user-defined derivative function (see *Mathematica* Help). One particular property of FreeQ is that it tests also heads of (sub)expressions, since it has an option Heads set to True by default (see the note on Heads options below).

As an example, imagine that we would like to define our own data type called Matrix, and a multiplication operation on it such that all the terms which do not contain < Matrix > head explicitly will be factored out, and commutative standard multiplication can be used on them, and inside our new multiplication command only the < Matrix > objects will remain. This is how we could do it :

```
Clear[ourTimes, Matrix];
ourTimes[a__, x__Matrix, b___] /; FreeQ[{a}, Matrix] :=
  Times[a, ourTimes[x, b]];
ourTimes[a___, x__Matrix, b__, c___] /; FreeQ[{b}, Matrix] :=
  Times[b, ourTimes[a, x, c]];
```

For instance :

```
ourTimes[a, Matrix[{1, 2}, {3, 4}], b, c,
 Matrix[{5, 6}, {7, 8}], d, Matrix[{9, 10}, {11, 12}]]
```

```
a b c d ourTimes[Matrix[{1, 2}, {3, 4}],
  Matrix[{5, 6}, {7, 8}], Matrix[{9, 10}, {11, 12}]]
```

4.3.7 A note on the Heads option

Many of the built - in *Mathematica* functions, and in particular all of the functions we just described, have an option Heads, which can be set as either Heads -> True or Heads -> False. For most functions (Position and FreeQ being exceptions), the default is Heads -> False. In this case, heads of expressions are excluded from searches and manipulations - they become "transparent" for these functions. However, the Heads -> True option makes them "visible" and then they are treated as any other expression. I will not go into further detail here, but let me just say that there are cases where this is important. Some examples can be found in *Mathematica* Help and *Mathematica* Book.

4.3.8 A more complicated example - finding subsequences

4.3.8.1 An example without an explicit pattern

Consider a following problem: we would like to know, if a combination <123> is encountered somewhere within first 500 digits of π. To solve this problem, we will resort to the RealDigits command. For example, the first 20 digits of π are given by

```
RealDigits[N[Pi, 20]][[1]]

{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4}
```

The solution to our problem then looks like this :

```
MemberQ[{RealDigits[N[Pi, 500]][[1]]}, {x___, 1, 2, 3, y___}]
False
```

How about the first thousand digits?

```
MemberQ[{RealDigits[N[Pi, 1000]][[1]]}, {x___, 1, 2, 3, y___}]
False
```

First two thousands?

```
MemberQ[{RealDigits[N[Pi, 2000]][[1]]}, {x___, 1, 2, 3, y___}]

True
```

The last answer is positive, but where exactly is this combination found? MemberQ does not answer this, and we need a bit more work :

```
Position[Partition[RealDigits[N[Pi, 2000]][[1]], 3, 1], {1, 2, 3}]
{{1925}}
```

To understand how it works, consider the following mini-example, cutting the list of digits to the first 20:

```
RealDigits[N[Pi, 20]][[1]]
Partition[RealDigits[N[Pi, 20]][[1]], 3, 1]
{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4}
{{3, 1, 4}, {1, 4, 1}, {4, 1, 5}, {1, 5, 9}, {5, 9, 2}, {9, 2, 6},
 {2, 6, 5}, {6, 5, 3}, {5, 3, 5}, {3, 5, 8}, {5, 8, 9}, {8, 9, 7},
 {9, 7, 9}, {7, 9, 3}, {9, 3, 2}, {3, 2, 3}, {2, 3, 8}, {3, 8, 4}}
```

We get a list of digits "sliced" in sublists of 3 with a shift of 1. Then, the Position command searches it for an element which is {1, 2, 3}, and returns its position. Obviously, the same position also is the position of < 1 > in 1, 2, 3 of the original list, since the "slicing" was done with just a unit shift (otherwise this would not be so). We can check this directly with Take :

```
digits = RealDigits[N[Pi, 2000]][[1]];
pos = First[First[Position[Partition[digits, 3, 1], {1, 2, 3}]]]
1925
```

Now we take the numbers starting at this position :

```
Take[digits, {pos, pos + 2}]
{1, 2, 3}
```

Let me make a general comment. On this example, we just saw a typical way to write and test a function in *Mathematica*: either before the function is written, or if the function has errors, the list with which it works is cut to a few first elements, and everything is "worked out" on this smaller list. What makes this non - trivial is that the list may be complex and nested (a tree) - this does not matter as long as the elements of the structure of the large real tree essential for the function operation are preserved in a small "test" tree.

Consider now 10000 digits:

```
Position[Partition[RealDigits[N[Pi, 10 000]][[1]], 3, 1], {1, 2, 3}] //
 Timing
```

```
{0.07, {{1925}, {2939}, {2977}, {3893}, {6549},
  {7146}, {8157}, {8773}, {8832}, {9451}, {9658}}}
```

These are all positions where < 123 > combination is encountered within a first 10000 digits of π.

- 4.3.8.2 An analogous example with patterns

As another example, let us find among the first 1000 digits of π all combinations of 3 digits which start with < 1 > and end with < 9 >, but the middle digit can be any. The code is as follows :

```
digits = RealDigits[N[Pi, 1000]][[1]];
partdigits = Partition[digits, 3, 1];
pos = Position[partdigits, {1, _, 9}]
Extract[partdigits, pos]
```

```
{{4}, {41}, {207}, {439}, {495}, {500}, {526},
 {548}, {705}, {713}, {731}, {923}, {985}, {998}}
```

```
{{1, 5, 9}, {1, 6, 9}, {1, 0, 9}, {1, 7, 9},
 {1, 1, 9}, {1, 2, 9}, {1, 3, 9}, {1, 7, 9}, {1, 9, 9},
 {1, 2, 9}, {1, 5, 9}, {1, 5, 9}, {1, 1, 9}, {1, 9, 9}}
```

Or

```
Cases[partdigits, {1, _, 9}]
```

```
{{1, 5, 9}, {1, 6, 9}, {1, 0, 9}, {1, 7, 9},
 {1, 1, 9}, {1, 2, 9}, {1, 3, 9}, {1, 7, 9}, {1, 9, 9},
 {1, 2, 9}, {1, 5, 9}, {1, 5, 9}, {1, 1, 9}, {1, 9, 9}}
```

It is more interesting to get all digit combinations together with their positions. The first thing that comes to mind is to combine together a list of digits and a list of their positions and then Transpose :

```
indexedcombs = Transpose[{Extract[partdigits, pos], pos}]
```

```
{{{1, 5, 9}, {4}}, {{1, 6, 9}, {41}},
 {{1, 0, 9}, {207}}, {{1, 7, 9}, {439}}, {{1, 1, 9}, {495}},
 {{1, 2, 9}, {500}}, {{1, 3, 9}, {526}}, {{1, 7, 9}, {548}},
 {{1, 9, 9}, {705}}, {{1, 2, 9}, {713}}, {{1, 5, 9}, {731}},
 {{1, 5, 9}, {923}}, {{1, 1, 9}, {985}}, {{1, 9, 9}, {998}}}
```

If one needs to remove the curly braces around the positions, this can be done with a simple rule:

```
indexedcombs /. {x_Integer} → x
{{{1, 5, 9}, 4}, {{1, 6, 9}, 41}, {{1, 0, 9}, 207}, {{1, 7, 9}, 439},
 {{1, 1, 9}, 495}, {{1, 2, 9}, 500}, {{1, 3, 9}, 526}, {{1, 7, 9}, 548},
 {{1, 9, 9}, 705}, {{1, 2, 9}, 713}, {{1, 5, 9}, 731},
 {{1, 5, 9}, 923}, {{1, 1, 9}, 985}, {{1, 9, 9}, 998}}
```

4.4 Functions - starting examples and syntax

■ 4.4.1 A definition and a simple example

By function we will mean a pair: any normal (non-atomic) *Mathematica* expression which contains patterns, and a rule in a global rule base, reflected by the DownValues command, which tells what should replace the first expression when it is encountered (we will ignore functions defined by SubValues, for the time being).

For example, this defines a function:

```
Clear[f, x];
f[x_] := x^3;
```

Here f[x_] is a normal expression (we see a Head <f> and single square brackets - the characteristics of the normal expression), the pattern is <x_> (we can see that it is a pattern by the presence of the underscore, which is one of the symbols that distinguish patterns; <x_> stands for exactly one argument), and one can check the presence of the global rule in a rule base by checking the DownValues command on a symbol <f> (see Chapter 1, section 1.2.3):

```
DownValues[f]
```

We can now make sure that the function works as intended :

```
{f[a], f[Pi], f[Einstein]}
```

$\{a^3, \pi^3, \text{Einstein}^3\}$

We see that the function as defined above works on any single expression.
By a single expression I mean single argument - for example <f> will not work in this situation:

```
f[a, b]
f[a, b]
```

This is not an error, but *Mathematica* simply does not know what to do with such an expression, and thus returns it back, in accordance with its general evaluation strategy.

Notice by the way, that all the ingredients needed to define a function we have already encountered before - patterns, assignment operator, etc. No new syntax is needed for function definitions - indeed because they are just some type of global rules, similar to variables. The non - trivial part here is in the action of an assignment operator (SetDelayed or sometimes Set): it decides whether or not the l.h.s. of the emerging global rule is legitimate, and if so, what type of global rule the new rule will be. When the l.h.s. contains patterns and the resulting global rule is stored in DownValues, we say that we have defined a function.

Often one needs to perform type or more general argument checks. They are very easy to implement in *Mathematica* and we will discuss them in the section on conditional patterns.

```
Clear[f];
```

4.4.2 More on function names and evaluation surprises

Consider a previous example:

```
Clear[f];
f[x_] := x^3;
```

Note that the straightforward attempt to check the Head (name of the function in this case) will give not what we would naively expect :

```
Head[f[t]]
Power
```

It is very easy to understand what happened by using the tracing command Trace:

```
Trace[Head[f[t]]]
```

$\{\{f[t], t^3\}, \text{Head}[t^3], \text{Power}\}$

We see that since the expression f[t] matches the pattern f[x_], the rule applied. Recall that the evaluation process by default starts from the leaves of the nested expression (from inside out - see section 2.5.6). Thus, when the Head command started to evaluate, its "content" has already changed from f[t] to t^3. The full internal form of t^3 is:

```
FullForm[t^3]
Power[t, 3]
```

This explains the end result. Going ahead of time, let me mention that there is a way to force the evaluation process to start in the opposite direction, from "branches" to "leaves" (non-standard evaluation , section 2.5.6), which will lead to the expected result for a function name:

```
Head[Unevaluated[f[t]]]
f
```

We have already discussed this construction in the section on variables (section 2.2.1).

```
Clear[f];
```

4.4.3 On the necessity of patterns

So, the name of the function is its Head - the symbol outside the square brackets in its definition, which contains a pattern. We may ask if it is possible to define a function without a pattern. The answer is that it is possible but the object so defined will not be a function in the normal sense and will have a behavior different from what we probably want. Here is an example:

```
Clear[f];
f[x] := x^3;
```

This definition does not contain a pattern (no underscore or other pattern ingredients). Let us check it:

```
{f[x], f[y]}
```

$\{x^3, f[y]\}$

Since we did not have a pattern, the class of expressions on which the corresponding rule will match has been narrowed down to just literal f[x]. In particular, it will not work on any other parameters :

```
{f[1], f[2], f[Pi], f[x]}
```

$\{f[1], f[2], f[\pi], x^3\}$

Moreover, if we then define the global value for an < x > variable, it will not work on < x > either :

```
x = 5;
f[x]
```

```
f[5]
```

(We should already be able to understand the last result : x evaluated to 5 before f had any chance to "look" at it).

The object f[x] here could be interpreted as an indexed variable (section 2.2.4) rather than a function, but even in this interpretation, it is a very error - prone practice to use symbols as indices in indexed variables. In any case, it has nothing to do with the behavior of the real function.

This behavior explains why we need patterns to define functions: patterns widen the class of expressions on which the rule will match. In particular, when we write

```
Clear[f, x];
f[x_] := x^3;
```

the pattern < x_ > means "any expression" and < x > here becomes a name attached to a placeholder where the actual input parameter will be placed. When we later call the function, normally the input parameters are evaluated first, and then the action of the function is to replace them with whatever the action of the r.h.s. of the corresponding rule should be.

```
Clear[f];
```

■ 4.4.4 More on the correct syntax of the function calls

Calling a function by name without the square brackets, or with parentheses used instead, will not give a desired result (and is a logical mistake in most cases):

```
{Sin, Sin (Pi)}
```

$\{\text{Sin}, \pi\,\text{Sin}\}$

In both cases, Sin was interpreted by *Mathematica* not as a function, but just as some symbolic object. In the latter case, parentheses were interpreted as a multiplication, which is easy to see with the help of FullForm :

```
FullForm[{Sin, Sin (Pi)} ]
List[Sin, Times[Pi, Sin]]
```

As we mentioned already, function calls are syntactically just a special case of *Mathematica* normal expressions, and thus have to obey the standard syntax rules. Thus, the single square brackets.

While just using a function name will not be a mistake in many languages (in C this will be a function pointer), in (strongly) typed languages this will lead to a type conflict and will probably cause a compiler warning or error message. Not so in *Mathematica*, which means that one has to be more careful. In version 6, the red highlighting will usually warn that the syntax may be wrong.

■ 4.4.5 On function definitions and assignment operators

■ 4.4.5.1 Use SetDelayed to define a function in most cases

What will happen, if we use the Set (=) command instead of SetDelayed (:=), when defining a function? This depends on the state of global variables present or defined in the system at the given moment. Here is an example:

```
Clear[f, x];
f[x_] = x^2;
```

```
{f[1], f[2], f[Pi], f[y]}
```

$\{1, 4, \pi^2, y^2\}$

The function works fine, but this is so only because by the moment of the definition, the variable < x > did not have any global value (no global rule was associated with it), and thus the r.h.s. x^2 evaluated trivially (to itself) and was recorded in the rule for function < f > in this way. This is what happens when < x > has a value at the moment of assignment :

```
Clear[f, x];
x = 5;
f[x_] = x^2;
```

```
{f[1], f[2], f[Pi], f[y]}


{25, 25, 25, 25}
```

To understand it better, we can look at DownValues of < f >, which reflect the way the definitions (rules) for < f > are stored in the system :

```
DownValues[f]
{HoldPattern[f[x_]] :→ 25}
{HoldPattern[f[x_]] :→ 25}
```

We see that now any input expression, regardless of its structure, will be replaced by 25. This behavior is in full agreement with the principles of operation of Set (=) assignment operator. It allows the r.h.s. of the definition to evaluate. This evaluation happens as usual, using the values for all global variables or expressions which exist in the system at the moment of the definition. Then Set uses the result of this evaluation as a r.h.s for the new global rule, associated with the l.h.s. of the assignment (See chapter 2 section 2.4.1). Since <x> had a global value 5, it was used in the calculation of the r.h.s, which then became the r.h.s. of the global rule associated with function <f> (definition of f).

So, the conclusion is that in the majority of cases functions must be defined with SetDelayed (:=) rather than Set (=). Since SetDelayed does not evaluate the r.h.s of an assignment, we are safe in this case.

■ 4.4.5.2 When Set is more appropriate

There are instances when Set operator is more appropriate do define a function however. In particular, this happens when a function may be symbolically precomputed so that it is stored in a form which allows a more efficient computation. Consider for instance a function defined as an indefinite integral, like the following one :

```
Integrate[Sqrt[1 + z^2], {z, 0, x}]
```

$$\frac{1}{2}\left(x\sqrt{1+x^2}+\text{ArcSinh}[x]\right)$$

```
Clear[g];
g[x_] := Integrate[Sqrt[1 + z^2], {z, 0, x}];
```

Let us compute it in a few points :

```
Table[g[i], {i, 10}] // Timing
```

$$\left\{7.791, \left\{\frac{1}{2}\left(\sqrt{2}+\text{ArcSinh}[1]\right), \sqrt{5}+\frac{\text{ArcSinh}[2]}{2}, \frac{1}{2}\left(3\sqrt{10}+\text{ArcSinh}[3]\right), \frac{1}{2}\left(4\sqrt{17}+\text{ArcSinh}[4]\right), \frac{1}{2}\left(5\sqrt{26}+\text{ArcSinh}[5]\right), \frac{1}{2}\left(6\sqrt{37}+\text{ArcSinh}[6]\right), \frac{1}{2}\left(35\sqrt{2}+\text{ArcSinh}[7]\right), \frac{1}{2}\left(8\sqrt{65}+\text{ArcSinh}[8]\right), \frac{1}{2}\left(9\sqrt{82}+\text{ArcSinh}[9]\right), 5\sqrt{101}+\frac{\text{ArcSinh}[10]}{2}\right\}\right\}$$

The point is, this integral can be computed in a closed form, and it absolutely makes sense to do it only once and then store the already computed definition. But with SetDelayed (as above), it will be recomputed every time afresh, according to a general rule of delayed evaluation. This is the case to use Set :

```
Clear[x, g1];
g1[x_] = Integrate[Sqrt[1 + z^2], {z, 0, x}];
```

The result is almost instantaneous this time (I cheated a bit by not including the time it took to compute the integral, but for a large number of function calls it will be in most cases negligible):

```
Table[g1[i], {i, 10}] // Short[#, 2] & // Timing
```

$$\left\{0., \left\{\frac{1}{2}\left(\sqrt{2}+\text{ArcSinh}[1]\right), \ll 8 \gg, \frac{1}{2}\left(10\sqrt{101}+\ll 1 \gg\right)\right\}\right\}$$

However, notice that we had to be careful and Clear the variable < x > . To be completely on the safe side, one can use one of the scoping constructs (discussed at the end of this chapter) to localize the variable :

```
Clear[g2];
Module[{x}, g2[x_] = Integrate[Sqrt[1 + z^2], {z, 0, x}]];
Table[g2[i], {i, 10}] // Short[#, 2] &
```

$\left\{\frac{1}{2}\left(\sqrt{2}+\text{ArcSinh}[1]\right), \ll 8 \gg, \frac{1}{2}\left(10\sqrt{101}+\ll 1 \gg\right)\right\}$

4.4.6 Assigning values to function symbols (names)

Since function symbols are just normal symbols, they can be used as variables and in particular can be assigned values. When the function is called on some argument, these values are computed before any other computation takes place. Consider an example:

```
Clear[f];
f[x_] := x^2;
f = Sin;
f[5]
```

```
Sin[5]
```

Notice that this does not mean that the previous rule for < f > disappeared - it is still in the rule base, as can be checked with DownValues :

```
DownValues[f]
```

$\{\text{HoldPattern}[f[x\_]] :\to x^2\}$

It is just that < f > now has also OwnValue < Sin >, which is computed in this case before any arguments are considered, and then the DownValue rule has no chance to apply :

```
OwnValues[f]
```

```
{HoldPattern[f] :→ Sin}
```

We can see what happens, with the help of the Trace command :

```
Trace[f[5]]
```

```
{{f, Sin}, Sin[5]}
```

To "restore" the function in this case, we obviously can not use Clear, since then also the DownValues of < f > will be cleared. In such a case, use Unset (section 2.2.6) :

```
f =.;
f[5]
```

```
25
```

In general, the above behavior means that one has to be careful and make sure that the symbol which is going to be used as a function name, does not have an OwnValue (unless this is what is desired, which is a rare case) - otherwise the newly defined function will not work properly.

■ 4.4.7 Advanced topic: parameter passing

■ 4.4.7.1 How parameters are passed

Let us look a bit closer at the way the parameters (which are the pattern tags and stand with blanks or other patterns on the l.h.s, such as x_) are passed to functions. The three main questions to address are these: what is the mechanism of parameter passing, is it possible to modify the passed parameters within a function such that the actual expressions being passed are modified after the function returns (pass-by-reference), and what are the rules for name collisions with the local variables. Since we did not systematically discuss the *Mathematica* scoping constructs yet, we will postpone the third question until such a discussion (sections 4.8, 4.10), and deal with the first two.

So, how are the parameters passed to the function? It turns out that the rule is very simple: their values (evaluated or not, depending on the presence of Hold attributes attached to the function) are textually substituted into the r.h.s of the function (before or after evaluation of the function itself takes place, again depending on the evaluation being standard or not). This happens somewhat similarly to the C preprocessor substitutions. What is important is that they never become local variables (in the sense of C), with the consequences we will describe in a second. We could say that the arguments are always passed by value, but the notion of value depends on whether or nor the function evaluates arguments in a standard way (presence or absence of Hold attributes).

The next question is whether or not the passed parameters can be modified inside a function. This depends on whether or not the passed object represents an L-value. The passed object will represent an L-value in 2 cases:

1. The evaluation order is standard, but what is passed evaluates (before being passed, according to the standard evaluation strategy that arguments are evaluated first) to a global symbol (which can be used as a variable in the sense described in section 2.2), with no assigned value.
2. What is passed is also a global symbol in the above sense, possibly with some global rule (definition) assigned to it, but the order of evaluation is non-standard and this symbol is passed unevaluated.

If the global symbol above is composite, and its head does not carry the Protected attribute, then the result of an assignment will be a DownValue or SubValue for the head.

In both of these cases it is possible to assign a value to a global symbol being passed to the function, from within the function, and thus modify it. Modification of the symbol in the first case has no direct analogs in languages such as C, just because it requires some symbol (which we pass) to hang around in a global name space but not have any value at all, which is only possible in a symbolic environment. In the second case, effectively the pass-by-reference semantics is simulated.

Finally, if what is passed does not represent an L - value, no assignments to it are possible. Again, this reflects the fact that what really happens is a textual substitution in a body of a function rather than say allocating variables on the stack. This textual substitution is similar to that performed by a scoping construct With (section 4.8.3).

Also, this means that there is no way of changing the value of the parameter locally (without global parameter modification) - either it represents an L-values and then is changed globally, or it does not and then no changes are at all possible. If one needs to change a passed parameter locally, one may introduce a local variable, initialize it with this parameter value, and then change a local variable instead (local variables will be described later in this chapter).

It is time now to illustrate the rather obscure statements I just made.

■ 4.4.7.2 Illustration: standard evaluation

We start with the following function which attempts to assign a value to the parameter passed to it :

```
Clear[f, x, a, b, c, d, h];
f[x_] := x = 5;
```

We start with a symbol which does not have a global value :

```
a
```

```
a
```

```
f[a];
a
```

```
5
```

We see that it was modified. This corresponds to the case 1 above. Now consider:

```
b = c;
f[b];
{b, c}
```

```
{5, 5}
```

It may be not immediately obvious, but what happened really is that only < c > received the numerical value, but not < b >, which has the same definition as before:

```
?b
```

```
Global‘b

b = c
```

In particular, if we now Clear[c], b will no longer evaluate to a number :

```
Clear[c];
b
```

```
c
```

What happened at the moment of the function call is that first, < b > evaluated to < c >, and then < c > was passed and subsequently modified, because it did not have any rule associated with it (if it had, then the r.h.s of this rule would be passed to the function or be evaluated further). Let us now repeat the first experiment again:

```
Clear[a];
f[a]
```

```
5
```

And call again :

```
f[a]
```

```
Set::setraw : Cannot assign to raw object 5. ≫
```

```
5
```

The point is that after the first call, the symbol < a > received a value < 5 >, to which it evaluated before being passed to the function. The function then attempted to assign 5 to 5, which does not work since the l.h.s. is not an L - value.

Consider now a different attempt :

```
Clear[a];
f[2 a]
```

```
Set::write : Tag Times in 2 a is Protected. ≫
```

```
5
```

Here, the object passed is < 2 a >, which is not an L - value either despite the presence of symbolic quantity < a > . It is easy to understand if we use the FullForm :

```
FullForm[2 a]
```

```
Times[2, a]
```

Therefore, what we really attempted to do was to define a rule for a built - in function Times, which is protected against this (if it weren't protected, the input would represent an L-Value, and the result of the assignment would be a DownValue rule for the head of the input; more on Protected attribute in section 4.9.5).

If we return to the second experiment :

```
Clear[b, c];
b = c;
f[b];
```

now we try again :

```
f[b]
```

```
Set::setraw : Cannot assign to raw object 5. ≫
```

```
5
```

We have the same story. The symbol < b > really acts as a middleman which simply hands < c > to the function. And the story with < c > is the same as what we had for < a > before.

Finally, let us consider the following input:

```
Clear[a];
f[h[a]];
```

Check :

```
h[a]
```

```
5
```

```
DownValues[h]
{HoldPattern[h[a]] :→ 5}
```

Although we have created a DownValue for < h >, < h > did not really become a function in the normal sense, since the l.h.s of the rule does not contain a pattern. Rather, we made a definition for a composite symbol < h[a] >, much like in our discussion in section 4.4.3.

- 4.4.7.3 Illustration: non-standard evaluation

Now we will modify our function to have a Hold attribute, which will mean that it will receive whatever argument we pass, in unevaluated form :

```
ClearAll[ff];
SetAttributes[ff, HoldFirst];
ff[x_] := x = 5;
```

We try now :

```
Clear[a];
ff[a];
a
5
```

And the second time :

```
ff[a];
a
5
```

We can modify < a > and call again :

```
a = 10;
ff[a];
a
5
```

So, the function really does modify the variable which has a global value. There is no mystery here: unevaluated simply means that the symbol < a >, rather than the r.h.s of the global rule associated with it, is passed to the body of the function, and thus modified. The symbol <a> here resembles the pointer to a variable in C.

How about our second experiment? We try:

```
Clear[b, c];
b = c;
ff[b];
{b, c}
{5, c}
```

We see that the result is completely different. Now < b > was assigned a value, and not < c > . But this had to be expected : unevaluated means in this case that the symbol < b >, rather than the r.h.s. of the rule associated with it (< c >), was textually substituted in the body of the function and thus modified. In particular, the previous definition < b = c > is lost now :

```
?b
```

```
Global`b

b = 5
```

And if we call ff[b] again, nothing will change. Finally, the calls with non-Lvalue objects will not work in this case too:

```
ff[Sin[c]]
```

```
Set::write : Tag Sin in Sin[c] is Protected. >>

5
```

Finally, let us return to our example with a composite object < h[a] > . It will turn out that in this case, the result of action of < ff > will be not so innocent, since it may alter a definition of a real function. Let us define a function:

```
Clear[h, a];
h[x_] := x^3;
```

Now we perform our manipulations :

```
a = 5;
ff[h[a]]
```

```
5
```

Let us look at the definitions of < h > now :

```
?h
```

```
Global`h

h[5] = 5

h[x_] := x^3
```

We did not cover it yet, but a function in *Mathematica* can have multiple definitions. What happened in this case is that a new specific definition has been added on the particular argument of < h> (5) , as a result of action of < ff > . You may wonder how comes that < a > inside < h > evaluated to < 5 > when we know that the argument is passed unevaluated to < ff > . In brief, < a > evaluated inside < h > before < ff > had any chance to "look" at the argument, because the question of whether or not < a > should be evaluated is decided by attributes of < h >, not < ff >. Had < h > have one of the Hold attributes, and < a > would not evaluate.

```
?h
```

```
Global‘h
```

```
h[5] = 5

h[x_] := x^3
```

Anyway, returning to the result, this is one good reason why it may become necessary to protect functions - while not many people will explicitly introduce erroneous rules for a function definition, they may sneak in as results of operations such as the one above. This is also one of the reasons why the programming style based on assignments, side effects and in-place modifications is not the preferred one in *Mathematica* programming - in a complex system such as *Mathematica*, with many more ways of doing things, this may result in all sorts of subtle bugs.

■ 4.4.7.4 Summary

So, to summarize: parameter passing is always in effect done by value through textual substitution, but the two circumstances make a variety of different behavior possible: first, symbols may be present in the system without any value attached to them, and second, the function may evaluate the argument (s) in standard or non - standard way. Whether or not the passed objects can be modified depends on whether they represent L-values at the moment of textual substitution, and it is completely equivalent to "hard-code" them in the form they have at this moment into the r.h.s and ask the same question for the resulting code.

This material can be somewhat unclear because we did not yet discuss in enough detail matters such as non-standard evaluation, Hold attributes and local variables. I recommend to revisit it after those topics are understood. There is nothing overly complicated in the parameter passing in *Mathematica* really, and on the other hand this topic is very important to understand.

```
ClearAll[f, ff, a, b, c];
```

■ 4.4.8 Function calls: prefix and postfix syntax

Apart from the standard way to apply a function to some expression, there exist two more short - hand forms for a function of a single argument: prefix and postfix notation. In the prefix notation, the special symbol < @> is used, while in the postfix notation, the double slash is used : < // > . For example :

```
Clear[f];
f[x_] := x^2;
{f[7], f@7, 7 // f}
```

```
{49, 49, 49}
```

The prefix form is convenient to remove extra square brackets when there is a piece of code with deeply nested function calls, like here :

```
{f[f[f[f[2]]]], f@f@f@f@2}
```

```
{65 536, 65 536}
```

The postfix notation is convenient when we want to apply some operation which is conceptually less important than the code it encloses, such as Timing measurements or rendering a matrix into a MatrixForm. In this way, it does not interfere with the main code when we read it:

```
IdentityMatrix[3] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Another reason to use the postfix notation is that the order in which the function calls are appearing corresponds to the order in which the transformations are applied to input, and this may make the code more readable in some cases.

One has to be careful with both prefix and postfix forms due to precedence however, as the following examples illustrate:

```
{f@x - y, f@x^y}
```

$\{x^2 - y, (x^2)^y\}$

In these cases, the result is such because the precedence of the subtraction or even Power operator is lower than that of the function call in the prefix notation. We have to use parentheses :

```
{f@(x - y), f@(x^y)}
```

$\{(x - y)^2, x^{2 y}\}$

Another example

```
matr = IdentityMatrix[3] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
Det[matr]
```

$$\text{Det}\left[\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\right]$$

The determinant has not been computed because the < matr > variable stores not just the matrix, but the matrix wrapped in a MatrixForm. This can be verified by looking at the FullForm :

```
FullForm[matr]
MatrixForm[List[List[1, 0, 0], List[0, 1, 0], List[0, 0, 1]]]
```

Once again, the parentheses must be used :

```
(matr = IdentityMatrix[3]) // MatrixForm
Det[matr]
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
1
```

Because of these precedence - related complications which often result in bugs, I would not recommend using these forms if this does not bring obvious advantages such as much improved code readability etc. Moreover, they are mostly used in interactive sessions and less so in complete stand-alone programs.

4.4.9 Function name conventions

I don’ t have much more to say here. Most of the rules which apply for variable names (section 2.2.1) also hold here. One difference worth mentioning is that a definition such as this:

```
Clear[g];
g[1][x_] := x^3
```

produces not the DownValue, but a SubValue for g :

```
{DownValues[g], SubValues[g]}
```

```
{{}, {HoldPattern[g[1][x_]] :> x^3}}
```

Whether or not to call this object a function is a matter of taste. In my definition above (and also below) I restrict functions to DownValues, but mostly because I didn’ t want to cover the SubValue case - it is not too often met in practice. On the other hand, I personally would consider the above defined < g > as good a function as any other.

Perhaps, one more comment - a stylistic one: as I already mentioned, it is a good practice to start the names of your symbols with a small letter, to avoid possible conflicts with the built - in symbols. But if the name contains only small letters, it is rather natural to interpret it as a name of a variable. One possibility to distinguish variables from functions is to use only lower - case letters for variables and a "camel notation" for functions - for example, the function to sum a list of numbers could be called < sumNumbers > . This is by no means standard, but I personally find it convenient, and in particular this is the style that I will use throughout the book.

4.5 Examples of functions of a single argument

All built-in functions (or commands) in *Mathematica* are functions in the sense described above: all of them have a format fname[arg1,...,argn] (caveat: sometimes, the symbol <fname> here may be not a symbol, but a normal expression itself, like for instance for an expression Derivative[1][f][x], which represents a first derivative of the function <f> with respect to the variable <x>, the <fname> symbol will actually be Derivative[1][f]. But such cases are not very frequent, and also represent no problems - they result in definitions stored in SubValues rather than DownValues).

The rules associated with the built-in functions can not normally be read - they are "wired in" the kernel (some of the externally-defined rules can be read, but they are also "hidden" by default). However, the way built-in functions work can be significantly altered by the user, since it is possible to redefine them and associate new rules with them, which will take precedence over the system-defined rules. All these techniques are not normally needed, fairly advanced and assume high level of competence with *Mathematica*. I mention these possibilities here just to illustrate the consistency of the whole approach. In principle, the built-in functions are not too different from the user-defined ones. They are just faster (being implemented in lower-level language like C), and already interconnected by a large base of global rules built into the system.

Let us now give some examples of functions of a single argument.

▪ 4.5.1 Example: Integer part of a number

Such a function exists in *Mathematica*, but we may define our own :

```
Clear[f];
f[x_] := IntegerPart[x];
```

```
{f[Pi], f[3 / 2], f[1]}
{3, 1, 1}
```

▪ 4.5.2 What we will not call a function definition

There is an alternative way of doing so :

```
Clear[g];
g = IntegerPart;
```

```
{f[Pi], f[3 / 2], f[1]}
{3, 1, 1}
```

The definitions such as a last one are not what we will call the definition of a function. Although the behavior of <f> and <g> look the same, there are subtle differences in how they are evaluated, which matter sometimes. For example, there are no DownValues associated with < g > :

```
DownValues[g]
{}
```

But rather, <g> has an OwnValue :

```
OwnValues[g]
{HoldPattern[g] :→ IntegerPart}
```

Although is most cases you will get what you want also by the second method, my advice is to avoid it until you get a good understanding of the evaluation process and the differences in evaluation induced by the differences in these two methods.

As an example where in fact the second method is more appropriate than the straightforwardly implemented first one, consider a built-in Increment function which has a side effect of incrementing a value of the input variable by 1:

```
Clear[a];
a = 5;
Increment[a];
a
6
```

We now define our version of an increment function :

```
Clear[ourInc];
ourInc[x_] := Increment[x];
```

Everything seems fine before we try it :

```
ourInc[a]
```

```
Increment::rvalue : 6 is not a variable with a value, so its value cannot be changed. ≫
```

```
6 ++
```

But if we define our function as

```
Clear[ourInc1];
ourInc1 = Increment;
```

Then :

```
ourInc1[a];
a
```

```
7
```

It works fine. We will save the detailed discussion of these issues for later chapters. Despite this example, in most cases the first method is the one to use, and in any case, the second one should not be really thought of as a method for defining a function. The closest analog I can think of is to create one more pointer to an already defined function in C - but this is not really a function definition. Let me just add that the above functionality of the Increment function can be also achieved with the first method employed with some modification (the code was actually given in Chapter 2, section 2.5.5).

We may now give a more formal distinction between what we will or will not consider a function definition (this is conventional) : w*e will say that the function with a name < f > is defined if the list of global rules returned by the DownValues[f] command is not empty*.

This definition is somewhat restrictive since it excludes for instance functions defined by SubValues and UpValues, but if you have real reasons to be uncomfortable with it, you perhaps shouldn't be reading this book (it takes a lot of experience to appreciate the cases missed by this definition).

4.5.3 Example: some trigonometric function

```
Clear[f];
f[x_] := Sin[Cos[Exp[x]]];
```

```
{f[1], f[1.]}
```

```
{Sin[Cos[ⅇ]], -0.790567}
```

4.5.4 Example: a function to reverse a string of symbols

```
Clear[f];
f[x_String] := StringReverse[x];
```

```
f["madam I am Adam"]
madA ma I madam
```

- 4.5.5 Example: A function of function

```
Clear[f, g];
f[g[x_]] := x * g[x];
```

```
{f[Sin[x]], f[g[y]], f[g[elephant]]}
{f[Sin[5]], y g[y], elephant g[elephant]}
```

This example is rather interesting. Since we did not "attach" a pattern to < g >, the rule will match when inside < f > we have literally < g > . However the argument of < g > can be anything, thanks to the pattern "attached" to the parameter x : < x_ > (in expression < x_ >, it is probably more correct to say that the tag < x > is attached to the pattern < _ >, since the presence and the form of the pattern plays a more fundamental role than the specific name of the input parameter). If we use a pattern (Blank[]) also with <g>, then also the function inside <f> can be arbitrary:

```
Clear[f, g, x];
f[g_[x_]] := x * g[x];
```

```
{f[Sin[x]], f[g[y]], f[g[elephant]]}
{x Sin[x], y g[y], elephant g[elephant]}
```

The pattern g_[x_] used here will match any expression of the form a[b]. Consider a more complicated example :

```
Clear[h, y];
h[x_] := 1 / x;
```

```
{f[h[y]], f[Unevaluated[h[y]]]}
```

$\left\{f\left[\frac{1}{y}\right], 1\right\}$

The Unevaluated command will be described in later chapters, but generally it forces the expression to be evaluated in a non - standard way (branches before leaves) at the particular point where Unevaluated is inserted. So, in the first case the evaluation started with the more innermost function h[y], which evaluated to 1/y, and it is this expression that the function < f > "saw". Since < f > does not have any rules associated with an expression 1/something, we got f[1/y] as a result. In the second case, the Unevaluated command forced the evaluation to start with <f>, and then the function <f> "saw" the expression h[y]. Since it has a corresponding rule, h[y] was replaced by y*h[y]. Then, h[y] was evaluated to 1/y, and then the final result was simplified to 1. All this evaluation dynamics that we just described can be seen by using the Trace command:

```
Trace[{f[h[y]], f[Unevaluated[h[y]]]}]
```

$$\left\{\left\{\left\{h[y], \frac{1}{y}, \frac{1}{y}\right\}, f\left[\frac{1}{y}\right]\right\},\right.$$
$$\left.\left\{f[h[y]], y\,h[y], \left\{h[y], \frac{1}{y}, \frac{1}{y}\right\}, \frac{y}{y}, \frac{y}{y}, 1\right\}, \left\{f\left[\frac{1}{y}\right], 1\right\}\right\}$$

```
Clear[f, g, h, x, y];
```

■ 4.5.6 Example: a function which exchanges another function and its argument

Consider another example: a function <f> will take another function <g> of argument <x>, g[x], and return x[g]:

```
Clear[f, g, h, x];
f[g_[x_]] := x[g];
```

```
{f[g[h]], f[f[g[h]]], f[x]}

{h[g], g[h], f[x]}
```

We see that applied twice, it returns back the original expression in the above case, and also that since there was no rule for <f> of an atomic argument, it returned just f[x]. However, the behavior may be different if there are rules for <g> and <h>:

```
Clear[f, g, h, x];
f[g_[x_]] := x[g];
h[g] := "Changed";
```

```
{g[h], f[g[h]], f[f[g[h]]], f[x]}

{g[h], Changed, f[Changed], f[x]}
```

In this case, the < f > applied twice will not return the original expression, again because it has changed before <f> had any chance to "see" its original form.

■ 4.5.7 Example: a recursive factorial function

Recursive functions are easy to implement in *Mathematica*. Here is for example the recursive factorial function:

```
Clear[fact];
fact[0] = 1;
fact[n_Integer ? Positive] := n * fact[n - 1];
```

Check :

```
{fact[0], fact[5], fact[-2]}
{1, 120, fact[-2]}
```

This example illustrates several points. First, it is possible for a function to have more than one definition on different arguments . In this case we had a separate definition for the base of the recursion, which can be checked also by looking at DownValues of < fact > :

```
DownValues[fact]
```

```
{HoldPattern[fact[0]] :→ 1,
 HoldPattern[fact[n_Integer ? Positive]] :→ n fact[n - 1]}
```

We see that two rules are present, not one.

Next, we see that by using a more complicated pattern (n_Integer?Positive in this case), we implemented a type-check, since the potentially dangerous input fact[-2] did not evaluate.

In the next sections, we will consider both aspects in more details: the patterns and rules associated with them, and functions with multiple definitions.

■ 4.5.8 Infinite iteration and recursion traps

In fact, in the symbolic rule-based environment like *Mathematica*, it is very easy to fall into an infinite recursion, for example like this (I have temporarily reduced the value of the system variable which controls the maximal iteration length, to its lower limit, but I would not recommend to try this if you have any unsaved results in your *Mathematica* session).

```
Clear[f, g, h, x];
f[x_] := g[x];
g[x_] := f[x];
h[x_] := h[x];
```

Now observe:

```
Block[{$IterationLimit = 20},
 f[x]]
```

```
$IterationLimit::itlim : Iteration limit of 20 exceeded. ≫
```

```
Hold[f[x]]
```

```
Block[{$IterationLimit = 20},
 h[x]]
```

```
$IterationLimit::itlim : Iteration limit of 20 exceeded. ≫
```

```
Hold[h[x]]
```

It is interesting that the above pathological function definitions result not in an infinite recursion (which would be the case in more traditional languages), but in infinite iteration. While I don't have an authoritative answer for why this is so, my guess is that this is probably due to the tail recursion being optimized in *Mathematica* (for tail-recursive functions, the recursion stack is not maintained since the result (recursive call) is always the last thing computed in such functions).

The following example is much more dangerous, and I don' t recommend to run it - just be aware of this kind of pitfalls.

```
Clear[f, x];
f[x_] := f[f[x]];
```

```
Block[{$RecursionLimit = 20, $IterationLimit = 20}, f[x]]
```

```
$RecursionLimit::reclim : Recursion depth of 20 exceeded. ≫

$RecursionLimit::reclim : Recursion depth of 20 exceeded. ≫

$RecursionLimit::reclim : Recursion depth of 20 exceeded. ≫

General::stop : Further output of $RecursionLimit::reclim will be suppressed during this calculation. ≫

$IterationLimit::itlim : Iteration limit of 20 exceeded. ≫

$IterationLimit::itlim : Iteration limit of 20 exceeded. ≫

$IterationLimit::itlim : Iteration limit of 20 exceeded. ≫

General::stop : Further output of $IterationLimit::itlim will be suppressed during this calculation. ≫
```

```
$Aborted
```

Here I had to manually abort the execution. Notice that here both infinite recursion and iteration took place, and even by limiting the corresponding "safety" system variables did not help much. In some of the cases similar to this one, one may need to kill the kernel and thus may loose all the unsaved results in a given *Mathematica* session.

- 4.5.9 An esoteric example: a self-destructive printing function

The following example will be rather extreme, and I am giving it to show that many things which are not possible in the more traditional programming environments, are possible in *Mathematica*. This example is a variation of the technique due to Michael Trott.

```
Clear[f, x, y, z];
z = 5;
f[x_] := (Clear[f]; Print[x]);
```

We check now :

```
Print[DownValues[f]];
f[z]
```

```
{HoldPattern[f[x_]] :→ (Clear[f]; Print[x])}

5
```

And again :

```
Print[DownValues[f]];
f[z]
```

```
{}
```

```
f[5]
```

What happened is that, once called, the function printed its argument, but also destroyed its own definition (deleted the rule associated with itself in the global rule base). When we called it second time, it did not have any rules associated with it any more, and thus returned "unevaluated". This would not change had we used it in a loop without user interruption:

```
Clear[f, x, y, z];
f[x_] := (Clear[f]; Print[x]);
Do[f[i], {i, 5}]
```

```
1
```

Only the first value was printed, since the function's definition disappeared after that.

This behavior is possible only because the delayed assignment (SetDelayed) was used in a function definition. This technique (or trick) can be quite useful is some cases, and also can be generalized, but this is outside of the scope of our present discussion.

■ 4.5.10 Mathematical functions and programming functions

The symbolic and rule - based nature of *Mathematica* removes the distinction between mathematical functions and functions in the sense of programming - the logically complete blocks of functionality. This may be quite confusing at the beginning, since we are used to the idea that these two types of functions are very different. As an illustration, we will now consider a function which will be a function in both senses at the same time :

```
Clear[f];
f[x_] := Sin[Print["This is a Sine function"]; x]
```

Now some examples :

```
f[5]
```

```
This is a Sine function
```

```
Sin[5]
```

```
f[Pi]
```

```
This is a Sine function
```

```
0
```

```
D[f[x], x]
```

```
This is a Sine function
```

```
Cos[x]
```

```
Solve[f[x] == 0, x]
```

```
This is a Sine function
```

```
Solve::ifun : Inverse functions are being used by Solve, so
    some solutions may not be found; use Reduce for complete solution information. ≫
```

```
{{x → 0}}
```

So, the distinction is really in our head. For *Mathematica*, it does not matter - both mathematical and programming functions at the end result just in some chains of rule applications.

4.6 Functions of several variables

So far, we considered in detail only functions of single argument. We will now consider functions of several variables.

4.6.1 Starting examples and a definition

Since we can define functions to work on lists, one way to define a function of several arguments is to define a function on a list of arguments. For example, if we need a function which raises one number to the power given by another number, we can do it as follows :

```
Clear[f, x, y];
f[x_List] := Power[x[[1]], x[[2]]];
```

```
{f[{2, 3}], f[{5, 2}], f[{10, 3}], f[3, 4]}
```

```
{8, 25, 1000, f[3, 4]}
```

In the last case the function did not evaluate, since its argument was not a list. This definition is however unsatisfactory for many reasons. First of all, the list of arguments in such a definition is non - uniform, since the first element gives the base while the second gives the power. This is not a good programming style and often leads to bugs in more complicated cases (There is nothing wrong in mixing elements of different types in a single list, but there should be more compelling reasons to do so). The second problem is that we have to impose an additional constraint that the length of the list is exactly two, otherwise we will get either errors or unexpected results:

```
{f[{1}], f[{2, 3, 4}]}
```

```
Part::partw : Part 2 of {1} does not exist. ≫
```

```
{1, 8}
```

Let us redefine :

```
Clear[f];
f[x_List /; Length[x] == 2] := Power[x[[1]], x[[2]]];
```

```
{f[{1}], f[{3, 4}], f[{2, 3, 4}]}
```

```
{f[{1}], 81, f[{2, 3, 4}]}
```

As a somewhat better alternative, we may define a function as follows:

```
Clear[f];
f[{base_, power_}] := Power[base, power];
```

```
{f[{1}], f[{3, 4}], f[{2, 3, 4}]}
```

```
{f[{1}], 81, f[{2, 3, 4}]}
```

This is not a bad way to do it, but then still there is no real necessity to combine arguments in a list.

Let us now make a definition : *the basic way to define a function of two argument is given by a construction* ***f[x_, y_] := r.h.s***.

For example :

```
Clear[f];
f[x_, y_] := Power[x, y];
```

Let us see :

```
{f[1], f[2, 3], f[5, 2], f[1, 2, 3], f[a, b], f[1.5, 2.0], f[E, Pi]}
```

$\{f[1],\ 8,\ 25,\ f[1, 2, 3],\ a^b,\ 2.25,\ e^{\pi}\}$

4.6.2 Putting constraints on the arguments

When defining functions, one can get much more from patterns by using constrained patterns (section 4.2.6). This allows to perform even rather complex argument checks as a part of the function definition on the left - hand side, rather than relegate the argument checks to the body of the function. This in turn leads to a much more readable and less error - prone code.

4.6.2.1 Constraints on separate arguments

We can impose some additional constraints on function arguments, using conditional patterns. For instance, we may require that our function works only on integers:

```
Clear[f];
f[x_Integer, y_Integer] := Power[x, y];
```

```
{f[1], f[2, 3], f[5, 2], f[1, 2, 3], f[a, b], f[1.5, 2.0], f[E, Pi]}
```

```
{f[1], 8, 25, f[1, 2, 3], f[a, b], f[1.5, 2.], f[e, π]}
```

We can make a weaker restriction and let our function work on any numbers :

```
Clear[f];
f[x_ ? NumberQ, y_ ? NumberQ] := Power[x, y];
```

```
{f[1], f[2, 3], f[5, 2], f[1, 2, 3], f[a, b], f[1.5, 2.0], f[E, Pi]}
```

```
{f[1], 8, 25, f[1, 2, 3], f[a, b], 2.25, f[e, π]}
```

We see that it did not evaluate on π, e. This is because the predicate NumberQ gives True only on explicit numbers. If we use a weaker yet predicate NumericQ (which gives True on any quantity on which the application of N command produces a number), we get:

```
Clear[f];
f[x_ ? NumericQ, y_ ? NumericQ] := Power[x, y];
```

```
{f[1], f[2, 3], f[5, 2], f[1, 2, 3], f[a, b], f[1.5, 2.0], f[E, Pi]}
```

```
{f[1], 8, 25, f[1, 2, 3], f[a, b], 2.25, ⅇ^π}
```

Of course, the conditions imposed on the arguments can be different for each argument. We can, for example, limit the base to be in the interval from 1 to 3:

```
Clear[f];
f[x_ /; (NumericQ[x] && 1 ≤ x ≤ 3), y_ ? NumericQ] := Power[x, y];
```

```
{f[1], f[2, 3], f[5, 2], f[1, 2, 3], f[a, b], f[1.5, 2.0], f[E, Pi]}
```

```
{f[1], 8, f[5, 2], f[1, 2, 3], f[a, b], 2.25, ⅇ^π}
```

Now the function evaluated non-trivially on the second, next to last and last expressions in the list.

- 4.6.2.2 Constraints that mix the function arguments

One can also impose more general constraints which will depend on both arguments. But in this case, the constraint has to be placed outside of the function parameters sequence, otherwise the function may not perform correctly (because it may use global values instead of those passed to the function, for some parameters. For a more detailed discussion of these issues, see *Mathematica* Help or *Mathematica* Book). Let us, for instance, define a function of 2 arguments, which will subtract the second from the first, but only if the first is equal to the square of the second:

```
Clear[g, a, b];
g[x_, y_] /; (x == y^2) := x - y;
```

```
{g[1, 2], g[9, 3], g[a, b], g[a^2, a]}
{g[1, 2], 6, g[a, b], -a + a^2}
```

If we want our function to work on numbers only, this can be done as before:

```
Clear[g, a, b];
g[x_ ? NumberQ, y_ ? NumberQ] /; (x == y^2) := x - y;
```

```
{g[1, 2], g[9, 3], g[a, b], g[a^2, a]}
{g[1, 2], 6, g[a, b], g[a^2, a]}
```

4.6.2.3 Using constraints to make functions safer

Here is another example: a function extracts a sublist of < n > elements from a list, but only if < n > is not larger than the length of the list :

```
Clear[grab];
grab[x_, n_] /; (n ≤ Length[x]) := Take[x, n];
```

```
{grab[{1, 2, 3, 4, 5}, 3], grab[{1, 2, 3, 4, 5}, 6]}
```

```
{{1, 2, 3}, grab[{1, 2, 3, 4, 5}, 6]}
```

If, for comparison, we just use Take, we get an error message in the second case :

```
{Take[{1, 2, 3, 4, 5}, 3], Take[{1, 2, 3, 4, 5}, 6]}
```

```
Take::take : Cannot take positions 1 through 6 in {1, 2, 3, 4, 5}. ≫
```

```
{{1, 2, 3}, Take[{1, 2, 3, 4, 5}, 6]}
```

However, our function is not completely foolproof, since the following call results in an error :

```
grab[{1, 2, 3, 4, 5}, 2.5]
```

```
Take::seqs : Sequence specification (+n, −n, {+n},
    {−n}, {m, n}, or {m, n, s}) expected at position 2 in Take[{1, 2, 3, 4, 5}, 2.5]. ≫
```

```
Take[{1, 2, 3, 4, 5}, 2.5]
```

It will be better to restrict our variables as follows :

```
Clear[grab];
grab[x_List, n_Integer] /; (n ≤ Length[x]) := Take[x, n];
```

```
grab[{1, 2, 3, 4, 5}, 2]
grab[1, 5]
grab[{1, 2, 3, 4, 5}, 2.5]
```

```
{1, 2}

grab[1, 5]

grab[{1, 2, 3, 4, 5}, 2.5]
```

Sometimes we may need some action to be performed in the case of incorrect input, for example some catchall error message to be issued or the error input analyzed for the error type. In this case, we may use a technique based on a possibility of *Mathematica* functions to have multiple definitions (to be discussed in the next session in detail). What we have to do is just to give our function another more general definition, like this:

```
Clear[grab];
grab[x_List, n_Integer] /; (n ≤ Length[x]) := Take[x, n];
grab[x__] := Print["Mistake in the type(s) of arguments"];
```

```
grab[{1, 2, 3, 4, 5}, 4]
grab[1, 5]
grab[{1, 2, 3, 4, 5}, 2.5]
```

```
{1, 2, 3, 4}
Mistake in the type(s) of arguments
Mistake in the type(s) of arguments
```

In the last definition we used the pattern with a double underscore (BlankSequence), to account for a case of 2 or more arguments, which is what we need here.

4.6.2.4 Warning: a subtle kind of mistakes

The matter of this subsection is likely to be obvious for many people, but it is important enough to be mentioned. When we build our functions on top of the built - in ones, we count, perhaps unconsciously, on the error - checking and warning messages of the built - in functions as a safety net. However, there could be cases when the input which is erroneous for us will be interpreted fine by the built - in function we use, but will perhaps mean something completely different from what we need. Consider the previous example with the following input :

```
grab[{1, 2, 3, 4, 5}, -3]
```

```
{3, 4, 5}
```

It is clear what happened: the built - in Take interpreted our input as to take the arguments from the end of the list. Did we really mean this functionality? May be, but may be not. To be absolutely safe, we had to use the <n_Integer?NonNegative> pattern. In particular, the following input already will result in an error message:

```
grab[{1, 2, 3, 4, 5}, -10]
```

```
Take::take : Cannot take positions −10 through −1 in {1, 2, 3, 4, 5}. ≫
Take[{1, 2, 3, 4, 5}, -10]
```

Generally, this means that extensive tests have to be performed on the code we write, since the absence of error messages on some input does not necessarily mean the correct logic.

```
Clear[f, g, grab];
```

4.6.3 Examples of functions of several variables (arguments)

Here I will give some examples of functions that take several arguments. Since I did not want to give built - in functions as examples, and on the other hand we don' t yet have the full functional machinery, some of the code may be unclear. You have two choices then: either have a quick look at the next chapter when this machinery is developed, or (recommended) just relax and consider it only an illustration of our present discussion.

4.6.3.1 Example: words containing a given substring

Here is a function which takes some list of words and returns all the words containing a given substring.

```
Clear[findWordsWith];
findWordsWith[textwords_List, str_String] :=
  Pick[textwords,
   Map[StringMatchQ[#, "*" <> str <> "*"] &, textwords]];
```

This will be our list of words (taken from *Mathematica* book)

```
wlist = {"Most", "of", "this", "Part", "assumes", "no",
   "specific", "prior", "knowledge", "of", "computer", "science",
   "Nevertheless", "some", "of", "it", "ventures", "into",
   "some", "fairly", "complicated", "issues", "You", "can",
   "probably", "ignore", "these", "issues", "unless", "they",
   "specifically", "affect", "programs", "you", "are", "writing"};
```

Some examples :

```
findWordsWith[wlist, "om"]
{computer, some, some, complicated}
findWordsWith[wlist, "ss"]
{assumes, Nevertheless, issues, issues, unless}
findWordsWith[wlist, "th"]
{this, Nevertheless, these, they}
```

In this example, we can make a function safer against bad inputs by checking that the incoming list is indeed a list of strings. This can be done rather easily with patterns :

```
Clear[findWordsWithSafer];
findWordsWithSafer[{textwords__String}, str_String] :=
  Pick[{textwords},
   Map[StringMatchQ[#, "*" <> str <> "*"] &, {textwords}]];
```

Since the check is purely syntactic, we should not expect a large performance overhead for this check. Notice that here we had to wrap <textwords> in a list inside the function since now it represents the sequence of strings - the interior of the list.

```
Clear[testtext, words, findWordsWith];
```

- #### 4.6.3.2 Example: transforming numbers to decimal from other bases

This function transforms a number given in an arbitrary base less than 10, into a decimal form.

```
Clear[convertToDecimal];
convertToDecimal[x_Integer, base_Integer /; base < 10] :=
  Fold[(base * #1 + #2) &, 0, IntegerDigits[x]];
```

For example :

```
convertToDecimal[10 001, 2]
17
convertToDecimal[10 001, 3]
82
```

Let us map our function on entire list (Map is a functional programming construct described in chapter V) :

```
binarylist = {100, 10, 1, 1011, 11 101};
convertToDecimal[#, 2] & /@ binarylist
{4, 2, 1, 11, 29}
Clear[binarylist, convertToDecimal];
```

- 4.6.3.3 Example: common digit subsequences of two numbers

Here we will be interested in finding common subsequences of digits of two integers. The function will take 3 arguments - numbers x and y, and the length of the common subsequence. It will return all common subsequences of this length, as a nested list.

```
Clear[commonSequences];
commonSequences[x_Integer, y_Integer, size_Integer] :=
  Intersection @@ (Partition[IntegerDigits[#], size, 1] & /@ {x, y});
```

Some examples :

Here are the numbers :

```
nm1 = 12 349 086 754 356 712 345;
nm2 = 12 378 695 435 348 712 356;
```

Here are their common subsequences of length 3, 4, and 5 :

```
commonSequences[nm1, nm2, 3]
{{1, 2, 3}, {3, 5, 6}, {4, 3, 5}, {5, 4, 3}, {7, 1, 2}}
commonSequences[nm1, nm2, 4]
{{5, 4, 3, 5}, {7, 1, 2, 3}}
commonSequences[nm1, nm2, 5]
{}
```

More serious example : random numbers 500 digits each.

```
(nm3 = Random[Integer, {1, 10^500}]) // Short
(nm4 = Random[Integer, {1, 10^500}]) // Short
```

```
271 458 083 351 ≪474≫ 6 872 373 986 376

272 528 102 705 ≪475≫ 0 739 006 022 299
```

Here are the common sequences :

```
commonSequences[nm3, nm4, 4]
```

```
{{0, 2, 4, 5}, {0, 2, 8, 9}, {1, 2, 8, 5}, {1, 4, 3, 6},
 {1, 8, 1, 7}, {2, 0, 8, 6}, {2, 3, 0, 7}, {2, 3, 9, 3},
 {2, 4, 0, 0}, {2, 5, 2, 8}, {3, 1, 0, 7}, {3, 5, 7, 9}, {4, 2, 7, 0},
 {4, 5, 8, 0}, {5, 0, 2, 0}, {5, 6, 1, 4}, {5, 6, 9, 5}, {6, 8, 2, 4},
 {6, 8, 6, 0}, {7, 0, 0, 2}, {7, 1, 8, 6}, {8, 1, 7, 1}, {8, 2, 4, 0},
 {8, 8, 9, 0}, {8, 9, 0, 1}, {9, 4, 4, 6}, {9, 4, 4, 9}, {9, 7, 3, 3}}
```

```
commonSequences[nm3, nm4, 5]
```

```
{{1, 8, 1, 7, 1}, {6, 8, 2, 4, 0}, {8, 2, 4, 0, 0}, {8, 8, 9, 0, 1}}
```

```
commonSequences[nm3, nm4, 6]
```

```
{{6, 8, 2, 4, 0, 0}}
```

```
commonSequences[nm3, nm4, 7]
```

```
{}
```

```
Clear[nm1, nm2, nm3, nm4, commonSequences];
```

■ 4.6.3.4* A longer example - numbers and intervals

This will be a more complicated example dealing with numbers and intervals. Please ignore the code in the body of the functions below, and concentrate on the patterns and type checks used in the definitions (l.h.s.) of these functions. If this example is still too hard, it can be skipped on the first reading.

Here are 20 random numbers in the range [1, 20] :

```
numbers = Table[Random[Integer, {1, 20}], {20}]
```

```
{18, 6, 12, 19, 7, 11, 8, 7, 17, 18, 11, 6, 13, 10, 16, 19, 13, 3, 6, 3}
```

Let us generate random (possibly overlapping) intervals :

```
nints = 6;
ints = Table[
  {x = Random[Integer, {1, 19}], x + Random[Integer, {1, 10}]}, {nints}]
```

```
{{5, 7}, {2, 7}, {13, 20}, {10, 15}, {14, 24}, {17, 21}}
```

The following function takes a list of numbers and a list of intervals, and returns intervals together with all the numbers which belong to them :

```
Clear[numsInIntervals];
numsInIntervals[nums_List, ints_List] :=
  MapAt[Sort, #, 2] & /@
   Reap[Function[{x}, Sow[x, Select[ints, #[[1]] ≤ x ≤ #[[2]] &]]] /@
      nums, _, List][[2]];
```

Let us check :

```
tst = numsInIntervals[numbers, ints]
```

```
{{{13, 20}, {13, 13, 16, 17, 18, 18, 19, 19}},
 {{14, 24}, {16, 17, 18, 18, 19, 19}},
 {{17, 21}, {17, 18, 18, 19, 19}}, {{5, 7}, {6, 6, 6, 7, 7}},
 {{2, 7}, {3, 3, 6, 6, 6, 7, 7}}, {{10, 15}, {10, 11, 11, 12, 13, 13}}}
```

The following function returns for a given number all the intervals which contain this number :

```
Clear[intervalsForNums1];
intervalsForNums1[nums_List, ints_List] :=
 Module[{x, y},
  Union /@ # & /@
    (Transpose /@ Outer[If[#2[[1]] ≤ #1 ≤ #2[[2]], {#1, #2},
          {#1, {}}] &, Union[nums], ints, 1]) /.
   {{x___, {}, y___} :> {x, y}, {x_Integer} :> x}]
```

Let us check :

```
intervalsForNums1[numbers, ints]
```

```
{{3, {{2, 7}}}, {6, {{2, 7}, {5, 7}}},
 {7, {{2, 7}, {5, 7}}}, {8, {}}, {10, {{10, 15}}},
 {11, {{10, 15}}}, {12, {{10, 15}}}, {13, {{10, 15}, {13, 20}}},
 {16, {{13, 20}, {14, 24}}}, {17, {{13, 20}, {14, 24}, {17, 21}}},
 {18, {{13, 20}, {14, 24}, {17, 21}}},
 {19, {{13, 20}, {14, 24}, {17, 21}}}}
```

Here is an alternative realization :

```
intervalsForNums2[nums_List, ints_List] :=
  Module[{y},
   y = Reap[
      Sow[#, Select[nums, Function[{x}, #[[1]] ≤ x ≤ #[[2]]]]] & /@
       ints, _, List][[2]];
   y = Join[y, {#, {}} & /@ Complement[nums, Transpose[y][[1]]]];
   Sort[y]];
```

Check again :

```
intervalsForNums2[numbers, ints]
{{3, {{2, 7}, {2, 7}}},
 {6, {{5, 7}, {5, 7}, {5, 7}, {2, 7}, {2, 7}, {2, 7}}},
 {7, {{5, 7}, {5, 7}, {2, 7}, {2, 7}}}, {8, {}},
 {10, {{10, 15}}}, {11, {{10, 15}, {10, 15}}}, {12, {{10, 15}}},
 {13, {{13, 20}, {13, 20}, {10, 15}, {10, 15}}},
 {16, {{13, 20}, {14, 24}}}, {17, {{13, 20}, {14, 24}, {17, 21}}},
 {18, {{13, 20}, {13, 20}, {14, 24}, {14, 24}, {17, 21}, {17, 21}}},
 {19, {{13, 20}, {13, 20}, {14, 24}, {14, 24}, {17, 21}, {17, 21}}}}
```

Both these functions can also serve as examples of code modularization (uses of Module construct) and functional programming. So far we give them just as examples of how typical user - defined functions of several arguments look like.

Type checking and function bulletproofing

For the last function, we can add conditions to check if the input is right. This will lead to a (small) performance overhead, though.

```
Clear[intervalsForNumsWithCheck];
intervalsForNumsWithCheck[nums_List, ints_List] /;
  And[And @@ (NumericQ /@ nums), Union[Length /@ ints] === {2},
   And @@ (NumericQ /@ Flatten[ints])] :=

 Module[{x, y},
  Union /@ # & /@
    (Transpose /@ Outer[If[#2[[1]] ≤ #1 ≤ #2[[2]], {#1, #2},
         {#1, {}}] &, Union[nums], ints, 1]) /.
   {{x___, {}, y___} :→ {x, y}, {x_Integer} :→ x}]
```

An additional condition checks that the list < nums > indeed contains only numeric quantities, that all sublists of the interval list < ints > have length 2 (that is, they indeed define intervals), and that all elements in the interval sublists are also numerical quantities. These conditions are implemented rather compactly using functional programming constructs. Note that we did not introduce any auxiliary functions which check these conditions - these functions actually "live" inside the condition check itself. Of course in cases when several large functions share the same condition checks, it may become advantageous to put these checks in separate functions.

Now that we looked at the way how these type checks can be implemented using functional programming constructs, we can consider a different way to implement them - based on patterns only :

```
Clear[intervalsForNumsWithCheckPattern];
intervalsForNumsWithCheckPattern[nums : {__ ? NumericQ},
  ints : {{_ ? NumericQ, _ ? NumericQ} ..}] :=
 Module[{x, y},
  Union /@ # & /@
    (Transpose /@ Outer[If[#2[[1]] ≤ #1 ≤ #2[[2]], {#1, #2},
         {#1, {}}] &, Union[nums], ints, 1]) /.
   {{x___, {}, y___} ⧴ {x, y}, {x_Integer} ⧴ x}]
```

We see that the pattern - based check is even more concise and elegant (and perhaps also faster in this case). We used a variety of pattern building blocks here: names for entire patterns (section 4.2.8), double underscore (BlankSequence, section 4.2.4.7), conditional patterns, repeated patterns (section 4.2.10).

Let us now check that our functions will not attempt to work on a wrong input.

```
intervalsForNumsWithCheck[
  {1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}}] // Short
```

```
{{1, {}}, ≪5≫, {7, {{3, 8}}}}
```

```
intervalsForNumsWithCheckPattern[
  {1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}}] // Short
```

```
{{1, {}}, ≪5≫, {7, {{3, 8}}}}
```

Now wrong inputs:

First attempt :

```
Clear[a];
intervalsForNumsWithCheck[{1, 2, 3, a, 5, 6, 7}, {{2, 6}, {3, 8}}]
```

```
intervalsForNumsWithCheck[{1, 2, 3, a, 5, 6, 7}, {{2, 6}, {3, 8}}]
```

```
intervalsForNumsWithCheckPattern[
 {1, 2, 3, a, 5, 6, 7}, {{2, 6}, {3, 8}}]
```

```
intervalsForNumsWithCheckPattern[
 {1, 2, 3, a, 5, 6, 7}, {{2, 6}, {3, 8}}]
```

Without protection :

```
intervalsForNums1[{1, 2, 3, a, 5, 6, 7}, {{2, 6}, {3, 8}}]
```

```
Transpose::nmtx : The first two levels of the one-dimensional
    list {If[2 ≤ a ≤ 6, {a, {2, 6}}, {a, {}}], If[≪1≫]} cannot be transposed. ≫

Transpose::nmtx : The first two levels of the one-dimensional
    list {If[2 ≤ a ≤ 6, {a, {2, 6}}, {a, {}}], If[≪1≫]} cannot be transposed. ≫

Transpose::nmtx : The first two levels of the one-dimensional
    list {If[2 ≤ a ≤ 6, {a, {2, 6}}, {a}], If[3 ≤ a ≤ 8, ≪1≫, {a}]} cannot be transposed. ≫

General::stop : Further output of Transpose::nmtx will be suppressed during this calculation. ≫

{{1, {}}, {2, {{2, 6}}}, {3, {{2, 6}, {3, 8}}}, {5, {{2, 6}, {3, 8}}},
 {6, {{2, 6}, {3, 8}}}, {7, {{3, 8}}}, Transpose[
  {If[2 ≤ a ≤ 6, {a, {2, 6}}, {a}], If[3 ≤ a ≤ 8, {a, {3, 8}}, {a}]}]}
```

Second attempt :

```
Clear[a];
intervalsForNumsWithCheck[{1, 2, 3, 4, 5, 6, 7}, {{2, a}, {3, 8}}]
intervalsForNumsWithCheck[{1, 2, 3, 4, 5, 6, 7}, {{2, a}, {3, 8}}]

intervalsForNumsWithCheckPattern[
 {1, 2, 3, 4, 5, 6, 7}, {{2, a}, {3, 8}}]
intervalsForNumsWithCheckPattern[
 {1, 2, 3, 4, 5, 6, 7}, {{2, a}, {3, 8}}]
```

Without protection :

```
intervalsForNums1[{1, 2, 3, 4, 5, 6, 7}, {{2, a}, {3, 8}}]
```

```
Transpose::nmtx :
  The first two levels of the one-dimensional list {If[2 ≤ a, {2, {2, a}}, {2, {}}], {2, {}}} cannot be transposed. ≫

Transpose::nmtx :
  The first two levels of the one-dimensional list {If[3 ≤ a, {3, {2, a}}, {3, {}}], {3, {3, 8}}} cannot be transposed. ≫

Transpose::nmtx :
  The first two levels of the one-dimensional list {If[4 ≤ a, {4, {2, a}}, {4, {}}], {4, {3, 8}}} cannot be transposed. ≫

General::stop : Further output of Transpose::nmtx will be suppressed during this calculation. ≫

{{1, {}}, Transpose[{If[2 ≤ a, {2, {2, a}}, {2}], {2}}],
 Transpose[{If[3 ≤ a, {3, {2, a}}, {3}], {3, {3, 8}}}],
 Transpose[{If[4 ≤ a, {4, {2, a}}, {4}], {4, {3, 8}}}],
 Transpose[{If[5 ≤ a, {5, {2, a}}, {5}], {5, {3, 8}}}],
 Transpose[{If[6 ≤ a, {6, {2, a}}, {6}], {6, {3, 8}}}],
 Transpose[{If[7 ≤ a, {7, {2, a}}, {7}], {7, {3, 8}}}]}
```

Third attempt :

```
intervalsForNumsWithCheck[{1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}, {1}}]
intervalsForNumsWithCheck[{1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}, {1}}]
```

```
intervalsForNumsWithCheckPattern[
 {1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}, {1}}]
intervalsForNumsWithCheckPattern[
 {1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}, {1}}]
```

Without protection :

```
intervalsForNums1[{1, 2, 3, 4, 5, 6, 7}, {{2, 6}, {3, 8}, {1}}]
Part::partw : Part 2 of {1} does not exist. >>
Part::partw : Part 2 of {1} does not exist. >>
Part::partw : Part 2 of {1} does not exist. >>
General::stop : Further output of Part::partw will be suppressed during this calculation. >>
Transpose::nmtx : The first two levels of the one−dimensional
    list {{1, {}}, {1, <<1>>}, If[1 ≤ {1}[[2]], {1, {1}}, {1, {}}]} cannot be transposed. >>
{Transpose[{If[1 ≤ 1[[2]], {1, 1}, {1}], {1}}],
 Transpose[{If[2 ≤ 1[[2]], {2, 1}, {2}], {2}, {2, {2, 6}}}],
 Transpose[{If[3 ≤ 1[[2]], {3, 1}, {3}], {3, {2, 6}}, {3, {3, 8}}}],
 Transpose[{If[4 ≤ 1[[2]], {4, 1}, {4}], {4, {2, 6}}, {4, {3, 8}}}],
 Transpose[{If[5 ≤ 1[[2]], {5, 1}, {5}], {5, {2, 6}}, {5, {3, 8}}}],
 Transpose[{If[6 ≤ 1[[2]], {6, 1}, {6}], {6, {2, 6}}, {6, {3, 8}}}],
 Transpose[{If[7 ≤ 1[[2]], {7, 1}, {7}], {7}, {7, {3, 8}}}]}
```

It is usually a good idea to protect your function from the wrong input. Of course, chances are that the wrong input uncaught by your code will trigger error messages of some built - in functions you are using, but if you want to build on the functionality you are presently developing, it is best to make your own functions acting as much like built - ins as possible. Once the quick-and-dirty solution is found and tested, it is a good practice to add input checks (just make sure that they are not redundant).

■ 4.6.4 Functions with variable number of arguments

Sometimes one may need to define a function whose number of arguments is not fixed, either because it can change from one call to the other, or because there is no need to refer to the individual arguments, or both. To do this, one may use BlankSequence (__) or BlankNullSequence (___), depending on whether or not the zero arguments case has to be included. Several built - in functions are in fact defined on an arbitrary number of arguments, examples being Plus, Times, Equal, Alternatives, SameQ and a few others.

As an example, we will define our own version of Plus function :

```
Clear[ourPlus];
ourPlus[x__] := Plus[x];
{ourPlus[1], ourPlus[1, 2], ourPlus[1, 2, 3]}
{1, 3, 6}
```

The built - in Plus is however defined also for zero arguments (through convention) :

```
Plus[]
0
```

Since we used BlankSequence, the zero argument case is not included in our function definition :

```
ourPlus[]
ourPlus[]
```

If we want it to behave just as the built - in Plus in all cases, we have to use BlankNullSequence :

```
Clear[ourPlus];
ourPlus[x___] := Plus[x];
```

Check now :

```
ourPlus[]
0
```

Let us now define a function which will multiply its first argument by the sum of all others :

```
Clear[firstMultiply];
firstMultiply[x_, y___] := x * Plus[y];
```

We have included the case when there is a single argument. Check :

```
{firstMultiply[1], firstMultiply[1, 2], firstMultiply[1, 2, 3]}
{0, 2, 5}
```

There is no ambiguity in this case as to which arguments will be matched with < x_ > pattern and which with < y___ >, since < x_ > states that there should be exactly one - the first one, and all the rest are matched by < y___ > .

The other way to define the same function would be not to split arguments on the level of the definition but to do that later, in the body of the function :

```
Clear[firstMultiplyAlt];
firstMultiplyAlt[x__] := First[{x}] * Apply[Plus, Rest[{x}]];
```

This is exactly equivalent to the previous realization. The role of Apply here is to hand to Plus the interior of the list Rest[{x}], rather than the list itself. :

```
{firstMultiplyAlt[1],
 firstMultiplyAlt[1, 2], firstMultiplyAlt[1, 2, 3]}
{0, 2, 5}
```

One has however to avoid ambiguous patterns like the following one :

```
Clear[f];
f[x__, y__] := Plus[x] * Plus[y];

{f[1, 2], f[1, 2, 3], f[1, 2, 3, 4]}
{2, 5, 9}
```

This example is in fact useful to examine the way how the pattern - matcher works: it is obvious that it starts from the left, and once it finds out that < 1 > matches < x__ >, then all the rest of the arguments match < y__ > . However, one should not rely on a particular behavior of the pattern - matcher, and neither should one use ambiguous patterns like the one above.

The more general pattern we use in a function definition, the more dangerous it is in the sense that it may match incorrectly in situations completely unforeseen by the developer. We already discussed this issue when we considered patterns and rules separately (section 4.2.4.7), but this is as true for function definitions as it was for local rules and patterns (since function definitions are just global rules). The less you use these patterns the safer your functions will be - use them only when necessary.

4.7 Functions with multiple definitions

We have seen such functions many time in our examples already, but here we will treat them more systematically. So, In *Mathematica* a function can have more than one definition. More precisely, there can be more than one rule associated with a function symbol, with different rules applying on different forms of arguments. In particular, one and the same function can be defined differently on different number and types of arguments. All this is possible because patterns are used for function definitions. To start with, consider an example:

4.7.1 Example: a discontinuous function

Consider a function which is 1 on integer numbers and - 1 on those which are not integer (in the spirit of the Dirichlet function - the latter is very different of course, being defined differently on rational and irrational numbers) :

```
Clear[f];
f[x_Integer] := 1;
f[x_] := -1;
```

We check :

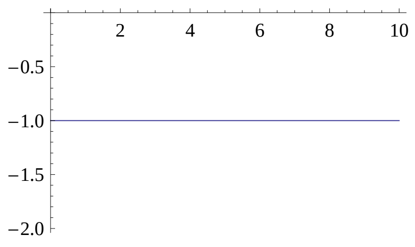
```
{f[1], f[1.5], f[2], f[2.5], f[4], f[Pi], f[E]}
```

```
{1, -1, 1, -1, 1, -1, -1}
```

Notice that in the second part of our definition in this case we don’ t necessarily need to use a pattern that is exact opposite of the first one, which would look like f[x_ /; Not[IntegerQ[x]]]. This is so because when the first pattern does not match, the second will match automatically, since it matches any single expression.

It is interesting that if we plot this function, the values on the integers (1) are not visible. One may think that this is because the probability that the sample point in the Plot procedure becomes exactly integer is very small (integers represent a set of measure 0). However, the truth is simpler: the numerical values for the sample points will never match the _Integer pattern just syntactically.

```
Plot[f[x], {x, 0, 10}]
```



■ 4.7.2 Adding more definitions

Let us now add another definition to our function, so that it will give 2 on every even number :

```
f[x_Integer ? EvenQ] := 2;
```

Check now :

```
{f[1], f[1.5], f[2], f[2.5], f[4], f[Pi], f[E]}
```

```
{1, -1, 1, -1, 1, -1, -1}
```

We see that nothing changed - it does not work (It is interesting that the result is correct in version 6). The reason can be seen by looking at function definitions:

```
? f
```

```
Global‘f

f[x_Integer] := 1

f[x_Integer?EvenQ] := 2

f[x_] := -1
```

By the way, the question mark in this context means the Information command, and returns the information contained in the global rule base on a given symbol (section 2.2.2).

We see that the reason for the above behavior is that *Mathematica* was able to figure out that the pattern f[x_Integer?EvenQ] is more specific than the pattern f[x_], but unable to figure out that it is also more specific than f[x_Integer] (this refers to versions prior to 6.0. In 6.0, the pattern-matcher does figure out the latter fact as well). The simplest thing one can do is to redefine the function, by placing definitions in a different order:

```
Clear[f];
f[x_Integer ? EvenQ] := 2;
f[x_Integer] := 1;
f[x_] := -1;
```

Check now :

```
{f[1], f[1.5], f[2], f[2.5], f[4], f[Pi], f[E]}
```

```
{1, -1, 2, -1, 2, -1, -1}
```

▪ 4.7.3 Changing definitions selectively

The above pattern - based mechanism of function definitions allows them to be very flexible. In particular, it is quite possible to change or delete a given definition corresponding to the specific pattern, without introducing changes in other definitions associated with this function.

To change an already existing definition for some pattern, to a new one, one just needs to redefine a function on this particular pattern with a new right hand side. For example, we want our first definition for < f > from the previous example to return not 2, but 4 on even numbers. We simply redefine :

```
f[x_Integer ? EvenQ] := 4;
```

Observe :

```
? f
```

```
Global‘f

f[x_Integer?EvenQ] := 4

f[x_Integer] := 1

f[x_] := -1
```

It is not required that the pattern tags (names) in a new pattern are literally the same as those for the old one (but otherwise the patterns have to be the same if we want to replace old definition with the new one) :

```
f[y_Integer ? EvenQ] := 6;
```

Check now :

```
? f
```

```
Global‘f

f[y_Integer?EvenQ] := 6

f[x_Integer] := 1

f[x_] := -1
```

As we see, the old definition still got replaced by a new one, since the pattern essentially did not change, and *Mathematica* can see that (this wasn’t the case in some early versions).

4.7.4 Warning: a common mistake

It is quite common during the development of some function to change the patterns for the function' s arguments. However, if one does not remove the old definition, it will remain in the rule base and may lead to errors when testing the function. Always make sure that you clear old definitions when you change a definition (argument patterns) of the function you are developing. One way to automate this is to always start with a line Clear[f] before any definition for < f > is entered - this is the practice I usually adhere to.

4.7.5 Selective removal of the definitions

If we would like to remove the definition of the function < f > associated with some pattern < pattern >, there is a special built - in command tailor - made for this : **Unset**. Its short - hand notation is <**= .**> (equal dot). Thus, we have to use either **f[pattern] =.**, or **Unset[f[pattern]]**. This will remove a given definition.

Let us for instance remove a first definition of the above function < f > . This is done as follows:

```
f[y_Integer ? EvenQ] =.
```

We now check :

```
?f
```

```
Global‘f
```

```
f[x_Integer] := 1

f[x_] := -1
```

As a side remark, it is interesting that the above possibilities of selective changes and/or removals of function definitions can be used in quite an unusual way : the function itself may (temporarily, for instance) change part of its own definitions during its execution. One reason why this may be useful is that sometimes it is a possible workaround to avoid an infinite recursion.

4.7.6 Case study: changing the weights of words

The problem

Consider some set of words, on pairs of which we will define a model "mutual attraction" function which will be equal to the number of common letters in the given pair of words. As a model set of words we will take the one we have used already :

```
wlist = ToLowerCase /@ {"Most", "of", "this", "Part", "assumes", "no",
    "specific", "prior", "knowledge", "of", "computer", "science",
    "Nevertheless", "some", "of", "it", "ventures", "into",
    "some", "fairly", "complicated", "issues", "You", "can",
    "probably", "ignore", "these", "issues", "unless", "they",
    "specifically", "affect", "programs", "you", "are", "writing"};
```

(we have converted words letters to the lowercase).

The solution

Our function will be a function of two strings - words. It is very easy to write - split words to characters, and compute a length of the intersection of the character lists:

```
Clear[wordFunction];
wordFunction[x_String, y_String] :=
  Length[Intersection[Characters[x], Characters[y]]];
```

For example :

```
wordFunction["word", "word"]
4
```

- Testing the solution

Let us now make a list of our words together with the weights that these words have with respect to some fixed word, say "computer" :

```
wlist1 = Table[{wlist[[i]], wordFunction["computer", wlist[[i]]]},
  {i, 1, Length[wlist]}]
{{most, 3}, {of, 1}, {this, 1}, {part, 3}, {assumes, 3}, {no, 1},
 {specific, 3}, {prior, 3}, {knowledge, 2}, {of, 1}, {computer, 8},
 {science, 2}, {nevertheless, 3}, {some, 3}, {of, 1}, {it, 1},
 {ventures, 4}, {into, 2}, {some, 3}, {fairly, 1}, {complicated, 6},
 {issues, 2}, {you, 2}, {can, 1}, {probably, 3}, {ignore, 3},
 {these, 2}, {issues, 2}, {unless, 2}, {they, 2}, {specifically, 3},
 {affect, 3}, {programs, 4}, {you, 2}, {are, 2}, {writing, 2}}
```

We can now sort the words according to the highest weight:

```
Sort[wlist1, #1[[2]] > #2[[2]] &]
{{computer, 8}, {complicated, 6}, {programs, 4}, {ventures, 4},
 {affect, 3}, {specifically, 3}, {ignore, 3}, {probably, 3},
 {some, 3}, {some, 3}, {nevertheless, 3}, {prior, 3}, {specific, 3},
 {assumes, 3}, {part, 3}, {most, 3}, {writing, 2}, {are, 2},
 {you, 2}, {they, 2}, {unless, 2}, {issues, 2}, {these, 2}, {you, 2},
 {issues, 2}, {into, 2}, {science, 2}, {knowledge, 2}, {can, 1},
 {fairly, 1}, {it, 1}, {of, 1}, {of, 1}, {no, 1}, {this, 1}, {of, 1}}
```

- Manipulating weights of individual words

Suppose now that we want to bring some words up in the list, that is, change the "strength function" of these words with the word "computer" by hand. Such new definitions can be implemented according to the above described scheme - we just have to add specific definitions of our weight function on specific words. Let these words be "programs", "knowledge" and "science". Let us give them weights :

```
wordFunction["computer", "programs"] = 20;
wordFunction["computer", "science"] = 15;
wordFunction["computer", "knowledge"] = 10;
```

Now let us have a look on the new definitions of < wordFunction > :

```
?wordFunction
```

```
Global`wordFunction

wordFunction[computer, knowledge] = 10

wordFunction[computer, programs] = 20

wordFunction[computer, science] = 15

wordFunction[x_String, y_String] := Length[(Characters[x]) ⋂ (Characters[y])]
```

Let us note two things: first, in these latter definitions we used Set rather than SetDelayed (it does not matter much for constant r.h.s.), and second, that these definitions are placed before the more general one even though they were added later - *Mathematica* figured out their level of generality and positioned them accordingly. This means that they will be applied before the more general one, and thus the general one does not "threaten" the more specific ones. Let us check now :

```
Sort[Table[{wlist[[i]], wordFunction["computer", wlist[[i]]]},
  {i, Length[wlist]}], #1[[2]] > #2[[2]] &]
```

```
{{programs, 20}, {science, 15}, {knowledge, 10}, {computer, 8},
 {complicated, 6}, {ventures, 4}, {affect, 3}, {specifically, 3},
 {ignore, 3}, {probably, 3}, {some, 3}, {some, 3}, {nevertheless, 3},
 {prior, 3}, {specific, 3}, {assumes, 3}, {part, 3}, {most, 3},
 {writing, 2}, {are, 2}, {you, 2}, {they, 2}, {unless, 2},
 {issues, 2}, {these, 2}, {you, 2}, {issues, 2}, {into, 2}, {can, 1},
 {fairly, 1}, {it, 1}, {of, 1}, {of, 1}, {no, 1}, {this, 1}, {of, 1}}
```

Now let us remove these definitions :

```
wordFunction["computer", "programs"] =.;
wordFunction["computer", "science"] =.;
wordFunction["computer", "knowledge"] =.;
```

We check now :

```
?wordFunction
```

```
Global`wordFunction

wordFunction[x_String, y_String] := Length[(Characters[x]) ⋂ (Characters[y])]
```

Only the general one remains.

- Automating the process (advanced)

It is interesting that the process of giving new definitions to some function can be automated by another function. In particular, let us define :

```
Clear[giveDefinitions];
giveDefinitions[f_, args_List, values_List] /;
   Length[args] == Length[values] :=
  (MapThread[Set, {Unevaluated[f[Sequence @@ #]] & /@ args,
     values}];);
```

This function is quite general (although one may write it more efficiently) : it takes the name of another function, a list of arguments and a list of values, and creates the new definitions for the supplied function accordingly. At the same time, any other definitions of this function will not be affected. This is our example :

```
giveDefinitions[wordFunction, {{"computer", "programs"},
  {"computer", "science"}, {"computer", "knowledge"}}, {20, 15, 10}]
```

We check now :

```
? wordFunction
```

```
Global`wordFunction

wordFunction[computer, knowledge] = 10

wordFunction[computer, programs] = 20

wordFunction[computer, science] = 15

wordFunction[x_String, y_String] := Length[(Characters[x]) ⋂ (Characters[y])]
```

The technique just illustrated allows some functions to manipulate the definitions of other functions, which allows us to control the program execution in a very flexible way.

Functions like < giveDefinitions >, which in effect manipulate other functions, are called higher - order functions. Their use is quite common in the functional programming style. We will cover a lot of built - in higher - order functions in the chapter V.

```
Clear[wordFunction, wlist, wlist1];
```

4.8 Larger functions, local variables and the code modularization

In the majority of real situations, the code for a typical function is longer than one or two lines (in other words, not every problem can be solved by one - liners). Also, it is often convenient to introduce intermediate variables, both to avoid redundant computations and to improve the code readability. Such variables one has to localize, in order to avoid name conflicts with the global variables already defined in the system, and in general not to "pollute" the global name space. On the scale of a single function or program, there are 3 constructs in *Mathematica* which provide this functionality: Module, Block and With. These constructs are explained in detail in *Mathematica* Book and *Mathematica* Help, so I will say just a few words about them here. On the larger scale, this is supported through the system of packages - we will consider them in part II.

▪ 4.8.1 Module

The goal of Module is to localize names of the variables, and avoid the name conflicts between the global names (and by global I mean everything exterior to the body of the Module), and the local names used in the code inside Module. What is important is that if this code calls some function which contains one of the global symbols with the name coinciding with the name of some of the local variables, the global value will be used. Put in another way, the variables are localized in space - only in the code inside Module, but not in functions which may be called from within this Module. The way Module does it is to create temporary variables with names which can not possibly collide with any other name (but see *Mathematica* Book for some subtleties). In fact, the workings of Module correspond most directly to standard variable scopes in other languages such as C.

The format of Module is **Module[{var1, var2, ...}, body]**, where var1, var2, ... are the variables we localize, and < body > is the body of the function. The value returned by Module is the value returned by the last operator in the <body> (unless an explicit Return[] statement is used within the body of Module. In this case, the argument of Return[arg] is returned). In particular, if one places the semicolon after this last operator, nothing (Null) is returned. As a variant, it is acceptable to initialize the local variables in the place of the declaration, with some global values : **Module[{var1 = value1, var2, ...}, body]**. However, one local variable (say, the one "just initialized" can not be used in the initialization of another local variable inside the declaration list. The following would be a mistake : *Module[{var1 = value1, var2 = var1, ...}, body]*. Moreover, this will not result in an error, but just the global value for the symbol <var1> would be used in this example for the <var2> initialization (this is even more dangerous since no error message is generated and thus we don’t see the problem). In this case, it would be better to do initialization in steps: ***Module[{var1=value1,var2,...}, var2=var1;body]***, that is, include the initialization of part of the variables in the body of Module.

One can use Return[value] statement to return a value from anywhere within the Module. In this case, the rest of the code (if any) inside Module is slipped, and the result <value> is returned.

One difference between Module and the localizing constructs in some other programming languages is that Module allows to define not just local variables, but local functions (essentially, this is because in *Mathematica* there is no strong distinction between the two). This opens new interesting possibilities, in particular this is useful for implementing recursive functions. The same comment holds also for the Block construct.

A simple example : here is a function which computes the sum of the first <n> natural numbers :

```
Clear[numberSum];
numberSum[n_Integer] :=
  Module[{sum = 0, i}, For[i = 1, i ≤ n, i ++, sum = sum +i]; sum];
```

```
numbersum[10]
```

```
numbersum[10]
```

```
{i, sum}
```

```
{i, sum}
```

■ 4.8.2 Block

Turning to the Block construct, it is used to localize the values of variables rather than names, or, to localize variables in time rather than in space. This means that in particular, if any function is called from within the Block (not being a part of the code inside this Block), and it refers globally to some of the variables with names matching those localized by Block, then the new (local) value for this variable will be used (this is in sharp contrast with Module). Block can be used to make the system temporarily "forget" the rules (definitions) associated with a given set of symbols.

The syntax of Block is similar to the one of Module. However, their uses are really different. While I will not go into further detail here (we will revisit scoping constructs in the part II), the quick summary is that it is usually more appropriate to use Module for localizing variables, and Block to temporarily change certain values. In particular, using Block instead of Module may result in errors in some cases. In general, if you use Block to localize a value of some variable, you have to make sure that no unforeseen variables with accidentally the same name will be involved in entire computation happening inside this Block, including possible (nested) calls of external functions which use these variables as global ones.

Here is some simple example with Block :

```
Clear[a, i];
a := i^2;
i = 3;
a
Block[{i = 5}, a]
```

```
9

25
```

We see that the value of < a > changed inside block, even though < a > was defined with the global < i > outside the Block, and no expilcit reference to < i > is present inside the Block.

It is worth mentioning that several built - in commands such as Do, Table, Sum and a few others, use Block internally to localize their iterator variables. This means that the same caution is needed also when one uses these commands as with the Block itself. We have already discussed this issue for Table (section 3.4.3) and Do (section 2.8.3).

4.8.3. With

The last scoping construct is With, and it is very different from both Block and Module. It is used to define local constants. With [{var1 = value1, ...}, body] is used to textually substitute the values < value1 > etc in every place in the < body > where < var1 >, etc occur. In some sense With is closer in spirit to the C preprocessor macros. In particular, it is not possible to modify the values given to the "variables" <var1> etc during the declaration, anywhere else inside With, since the occurrences of <var1> etc are textually substituted with the values <val1> etc before any evaluation takes place. For example, the following code:

```
With[{i = 2}, i = 3]
```

is just equivalent to a direct attempt of assigning the value 3 to 2 :

```
With[{i = 2}, i = 3] // Trace
```

Set::setraw : Cannot assign to raw object 2. ≫

```
{With[{i = 2}, i = 3], 2 = 3, {Message[Set::setraw, 2],
  {Set::setraw, Cannot assign to raw object ‘1‘.},
  {MakeBoxes[Set::setraw : Cannot assign to raw object 2. ≫,
   StandardForm], RowBox[{RowBox[{Set, ::, "setraw"}], : ,
    "Cannot assign to raw object \!\(2\). \!\(\*ButtonBox[\"≫\",
     ButtonStyle->\"Link\", ButtonFrame->None,
     ButtonData:>\"paclet:ref/message/Set/setraw\",
     ButtonNote -> \"Set::setraw\"]\)"}]}, Null}, 3}
```

The With construct is very useful in many circumstances, particularly when some symbols will have a constant value throughout the execution of some piece of code. Since these values can not be changed once initialized by With, it improves the code readability because it is easy to find the place where the symbols are defined, and then we know that they will not change. There are also more advanced applications of With, some of which we will discuss later (for example, one such application is to embed parameters into functions which are created at run-time).

The scoping constructs Block, Module and With can be nested arbitrarily deep one within another. Possible name conflicts are resolved typically in such a way that the more "internal" definitions have higher priority. *Mathematica* Book contains a lucid discussion of the subtleties associated with name conflicts in nested scoping constructs.

4.9 Function attributes

Apart from the definitions, functions can be assigned certain properties which affect the way they are executed. These properties are called Attributes. There are many possible attributes which a function may have, and we will only briefly discuss very few of them here. It is important that all possible attributes are only those built in *Mathematica*, and one can not assign to a function a "home-made" attribute that *Mathematica* does not know.

4.9.1 Listable attribute and SetAttributes command

4.9.1.1. A simple example

This attribute is used when we want our function to be automatically threaded over any lists passed to it as arguments. For example, let us define a function which will square its argument and will also work on lists :

```
Clear[flst];
flst[x_] := x^2;
SetAttributes[flst, Listable];
```

Notice how we set the attributes: we use the **SetAttributes** built-in function. Let us check :

```
testlist = Range[10]
testlist1 = Range /@ Range[5]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}}
```

```
flst[testlist]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
flst[testlist1]
```

```
{{1}, {1, 4}, {1, 4, 9}, {1, 4, 9, 16}, {1, 4, 9, 16, 25}}
```

4.9.1.2 Careful with the Listable attribute

As we can see, listability leads to function working also on nested lists. In fact, this is not always the desired behavior. For example, here we have a function that takes an interval and computes its length:

```
Clear[intervalLength];
intervalLength[{start_, end_}] := end - start;
```

We now want it to work on a list of intervals and add the Listable attribute :

```
SetAttributes[intervalLength, Listable];
```

Now we use it on a list of intervals :

```
intervalLength[{{1, 4}, {2, 7}, {5, 10}}]
```

```
{{intervalLength[1], intervalLength[4]},
 {intervalLength[2], intervalLength[7]},
 {intervalLength[5], intervalLength[10]}}
```

We see that listability made our function go all the way to the elements which are not lists - but this is not we want. So, if you attach a listable attribute to a function, make sure that its normal arguments are not lists.

4.9.1.3 A way out in some cases

As another example of the similar kind, consider a following one: we are given some function , say < f >, and two lists , say {1, 2} and {3, 4, 5}, and wish the output be a list : {f[1, {3, 4, 5}], f[2, {3, 4, 5}]} - that is, to thread < f > over the first list but not the second. For the same reason as above, the straightforward attempt to assign a listable attribute to < f > will fail :

```
ClearAll[f];
SetAttributes[f, Listable];
f[{1, 2}, {3, 4, 5}]
```

Thread::tdlen : Objects of unequal length in f[{1, 2}, {3, 4, 5}] cannot be combined. ≫

```
f[{1, 2}, {3, 4, 5}]
```

I can’ t help showing here a hack which solves this sort of problems and which is related to the use of Listable SubValues, although it is perhaps a bit too advanced at this point.The idea is that we will create a higher - order function which will take < f >, and both of our lists as parameters. Here is the code :

```
ClearAll[listThread];
listThread[f_, x_, y_] := Module[{auxf},
   Attributes[auxf] = {Listable};
   auxf[t_][z_] := f[t, z];
   Through[auxf[x][y]]];
```

Check :

```
ClearAll[f];
listThread[f, {1, 2}, {3, 4, 5}]
```

```
{f[1, {3, 4, 5}], f[2, {3, 4, 5}]}
```

What happens here is that an auxiliary function is defined inside Module, but if you look carefully at its definition you will realize that it corresponds to global rules stored in SubValues rather than DownValues (section 2.2.5), because the function <auxf[x]>, considered as a function of <y>, has a composite (non-atomic) head. Setting the Listable attribute to < auxf > will then only affect the "first" argument < x >, but not < y > . Note also that neither <t> nor <z> needs to be localized since SetDelayed is used in the definition, and thus they are local to the auxiliary function scope automatically. The Through operator is needed here as well - it is covered at the end of chapter V.

This trick is trivial to generalize to the total <n> number of arguments, out of which you need your function to be Listable on <k>: just place these <k> first - in the place of our <t>, and the rest - in the place of <z>: auxf[arg1...argk][arg(k+1)...argn].

4.1.9.4 Be aware of Listable built - in functions.

There are at least two good reasons to check for a Listable attribute of a built - in function you wish to use.

First - to avoid errors of the type described above, which result from the assumption that the function is not Listable when in fact it is. A classic example here would be an attempt to sum two nested lists of the same length, but where lengths of sublists in the two lists are different:

```
{{1, 2}, {3, 4, 5}, {6}} + {{1}, {2, 3}, {4, 5, 6}}
```

```
Thread::tdlen : Objects of unequal length in {1, 2} + {1} cannot be combined. ≫

Thread::tdlen : Objects of unequal length in {3, 4, 5} + {2, 3} cannot be combined. ≫

Thread::tdlen : Objects of unequal length in {6} + {4, 5, 6} cannot be combined. ≫

General::stop : Further output of Thread::tdlen will be suppressed during this calculation. ≫

{{1} + {1, 2}, {2, 3} + {3, 4, 5}, {6} + {4, 5, 6}}
```

This result is such (error messages) because summation operator Plus is Listable. For the record, Listable attributes, among others, can be removed or temporarily disabled to avoid problems like this, for both user-defined and built-in functions. We will see such an example in chapter V.

The second reason to be aware of Listable attributes for built-ins is to be able to write more efficient code. If some built - in function is Listable and one has to thread it over a list, it will almost certainly be faster to feed it an entire list rather than to thread (map) it by hand with commands such as Table or Map. This is so just because more operations will then be "pushed" into the kernel. For user - defined functions however there will be no significant difference in most cases, so this comment refers to built - ins.

As an example, consider computing some function numerically on a list of first 50000 natural numbers. Here is implementation using Table :

```
Table[N[Exp[Sin[i]^3]], {i, 1, 50 000}] // Short[#, 3] & // Timing
{0.07, {1.81452, <<49 998>>, 0.368056}}
```

Here we use Listability of all the functions (Sin, Exp, Power) to compute the result on entire list. We win a factor of 7 - 10 (an order of magnitude) in performance.

```
Exp[Sin[N[Range[50 000]]]^3] // Short[#, 3] & // Timing
{0.01, {1.81452, <<49 998>>, 0.368056}}
```

■ 4.9.2 Clearing Attributes - the ClearAll command

Now suppose we would like to give our function < flst > from the previous example another definition, and also no longer want it to have a Listable attribute (in fact, we want to remove all attributes attached to the symbol < flst >). First thing we may try is just to use Clear command, as we usually do :

```
Clear[flst];
flst[{1, 2, 3, 4, 5}]
{flst[1], flst[2], flst[3], flst[4], flst[5]}
```

We see that while the definition of < flst > has been cleared, the Listable attribute remains. To remove both the definitions and the attributes attached to a given symbol, use ClearAll instead of Clear :

```
ClearAll[flst];
flst[{1, 2, 3, 4, 5}]
flst[{1, 2, 3, 4, 5}]
```

Let me stress that ClearAll serves to clear all definitions (including attributes) for a given symbol (or symbols), and not to clear definitions of all global symbols in the system (it is a common mistake to mix these two things).

4.9.3 Orderless attribute

This attribute states that the result of evaluation of a given function should not depend on the order of its arguments, which is commutativity. The presence of this attribute does change the evaluation of the function, because then the argument list is sorted (by default *Mathematica* sorting function) before the actual evaluation process for this function starts. Many built-in functions such as Plus or Times (or, in general, commutative functions) have this attribute. As an example, we can arrange sorting of a list (with the default sorting criteria) by just defining a "container function" with such an attribute:

```
ClearAll[fsort];
SetAttributes[fsort, Orderless];
```

```
testlist = Table[Random[Integer, {1, 15}], {20}]
{13, 13, 15, 11, 10, 6, 5, 13, 3, 5, 14, 8, 8, 14, 7, 14, 10, 10, 1, 11}
Apply[fsort, testlist]
fsort[1, 3, 5, 5, 6, 7, 8, 8, 10,
 10, 10, 11, 11, 13, 13, 13, 14, 14, 14, 15]
```

The meaning of Apply will be clarified in the chapter V. Its role here is to "eat up" the List head so that the < fsort > receives a sequence of arguments rather than a list.

4.9.4 Flat attribute

This attribute is used to implement associativity. This means that for example expression like f[a,b,f[c,d,e,f[f[g,h]]],i,f[f[j]]] will be automatically simplified to f[a,b,c,d,e,f,g,h,i,j] if the symbol <f> has a Flat attribute. Previously we considered a rule-based way to mimic this functionality in a very special case when the function has a property that f[f[f[...f[x]]]]] = f[x]. With a Flat attribute this is trivial since the system does all the work. For instance:

```
ClearAll[f, x];
```

```
testlist = NestList[f, x, 5]
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]], f[f[f[f[f[x]]]]]}
```

Now we set the Flat attribute to the function < f > :

```
SetAttributes[f, Flat];
```

```
testlist
{x, f[x], f[x], f[x], f[x], f[x]}
```

And our first example :

```
Clear[a, b, c, d, e, g, h, i, j];
f[a, b, f[c, d, e, f[f[g, h]]], i, f[f[j]]]
f[a, b, c, d, e, g, h, i, j]
```

By the way, setting the attributes is largely independent from giving definitions to a function. The non-trivial dependencies arise in some cases, and generally one has to set up attributes before any definitions are given to the function. However, often there is no need to satisfy such strict requirements (but you have to know precisely what you are doing, of course). In particular, some attributes may be set when the function has already been defined for a while and perhaps used, attributes may also be set temporarily, or selectively removed. In fact, as an extreme case, a function may be programmed in such a way that it itself temporarily removes, changes or restores its own attributes (this is however a really exotic example).To remove a given attribute, one has to use **ClearAttributes**. The current list of attributes can be monitored with the Attributes built - in command :

```
Attributes[Plus]
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

■ 4.9.5 Protected attribute

This attribute is needed if we want to protect a given function or symbol against changes that the user or some user program may wish to apply to it. Most system functions have the Protected attribute. For example, when we try something like this assignment:

```
a + b = c
Set::write : Tag Plus in a + b is Protected. >>
c
```

The FullForm Plus[a, b] = c tells us that we are trying to make a new rule (definition) for the built - in Plus command, which is protected. Of course , our assignment fails.

It is possible to make a symbol Protected by using the built - in command Protect and to unprotect the symbol by using the built - in command Unprotect. This is often handy. Protecting your own symbols is a standard practice when writing packages (which are system extensions to some domain), while unprotect-ing is used usually with built - in commands when we need to add some new rule to the definition of this or that built - in command. As an example, we may Unprotect Plus command so that the above assign-ment will work :

```
Clear[a, b, c];
Unprotect[Plus];
Plus[a, b] = c;
Protect[Plus];
```

We can check now :

```
a + b
c
```

The reason we can redefine the behavior of system functions is that the user - defined rules have higher priority than the system ones. But what we just did was in this case not motivated by any serious need and thus represents an act of vandalism. Besides, even in cases when the workings of the built-in functions have to be modified, modifying their DownValues (adding rules as above) is really a last resort. There are softer ways of getting what one needs, such as using UpValues for the symbol you define. I will have more to say about this later. For now, then, let us remove our definition :

```
Unprotect[Plus];
Clear[Plus];
Protect[Plus];
```

```
a + b
a + b
```

4.9.6 Attributes are properties of symbols

I would like to stress that while we may interpret many attributes to be properties of functions, they are really properties of symbols (function names for functions). Function definitions are rules associated also with symbols (function heads or names). There is no fundamental distinction between rules describing functions and just some symbolic rewritings, as we have already discussed a few times. The technical distinction is that say rules for symbols are kept as OwnValues and rules for functions in DownValues (and UpValues and SubValues which we did not cover yet), but the main point is the same: there are symbols and associated with them global rules and properties. Whether we interpret these symbols as function names or something else is up to us.

4.9.7 Attributes HoldFirst, HoldRest and HoldAll

4.9.7.1 The meaning of argument holding

These attributes are used when some of the function arguments have to be evaluated only after the rules associated with the function name have been applied. This means that these attributes change the evaluation order from standard evaluation (depth-first, subexpressions before expressions) to a non-standard one (expressions before subexpressions). One usually needs to change the evaluation order to do something non-trivial. In particular, as we have seen already on the example of the increment function <inc[x]> (sections 2.5.5, 4.5.2), Hold attributes can be used to mimic the pass-by-reference semantics. This allows functions to modify the variables which are passed to them. Other cases when one needs to hold some arguments unevaluated arise when only some of the arguments have to be evaluated at all, and which ones have to be evaluated is decided by say a condition on the part of arguments that are evaluated (this is exactly the situation with conditional operators such as If).

The attribute HoldFirst instructs a function to hold (in unevaluated form) the first argument. HoldRest instructs to hold all but the first argument, and HoldAll instructs to hold all arguments. The fact that the argument is held unevaluated does not necessarily mean that it is never evaluated in a function (which may also happen if it is discarded before it is evaluated) - it simply means that it is evaluated after all the transformations of this argument by the function <f> (according to the definition of <f>) are performed. As a simple example, consider a squaring function:

```
ClearAll[f];
f[x_] := x^2;
```

Let us Trace its evaluation on some number :

```
Clear[a];
a = 5;
f[a] // Trace
```

```
{{a, 5}, f[5], 5^2, 25}
```

We see that < a > was evaluated before < f > . Now let us attach the HoldFirst attribute to < f > :

```
SetAttributes[f, HoldFirst];
```

Now :

```
f[a] // Trace
```

```
{f[a], a^2, {a, 5}, 5^2, 25}
```

We see that now the evaluation order has changed : first the function < f > was evaluated, and then the value of < a > was substituted. In this simple example, the end result was the same regardless of the evaluation order, but in less trivial cases the evaluation order becomes important.

It is fairly easy to give an example of held arguments being discarded and thus not evaluated at all - take any operators on the False branch of some If operator.

- 4.9.7.2 Advanced topic: Hold attributes and pattern-matching

While the general topic of Hold attributes is a bit too advanced for us now (since it requires a much more thorough discussion of the evaluation process), let me mention one important point. This is, Hold attributes affect pattern-matching. Consider the following function.

```
ClearAll[f];
f[x_Sin] := x^2;
f[x_] := "Not sine"
```

It is supposed to square any expression of the form Sin[anything], and issue a message for all other inputs. We can try it:

```
Clear[a, b];
{f[Sin[a]], f[a], f[Sin[Pi]]}
```

```
{Sin[a]^2, Not sine, Not sine}
```

In the last input, Sin[Pi] was evaluated first, leading to f[0], which led to a "Not sine" message. Let us now add the attribute:

```
SetAttributes[f, HoldFirst];
```

And test the same input again:

```
Clear[a, b];
{f[Sin[a]], f[a], f[Sin[Pi]]}
```

```
{Sin[a]^2, Not sine, 0}
```

What happened with the last output is that the presence of Hold attribute made a function to evaluate "branches before leaves", and then it had a chance to "see" Sin[Pi] before it evaluated to 0, and thus the first definition applied.

All right, this is all known stuff, we discussed the non-standard evaluation before. But now, let us do it a bit differently :

```
Clear[a, b];
a = Sin[b];
{a, f[a]}
```

```
{Sin[b], Not sine}
```

For us, it is obvious that <a> is Sin[b], so this behavior looks like a bug. It isn't however: Hold attribute means that the argument is held unevaluated before the rules associated with the function apply. If we supply the direct Sin[something], then, while Sin[something] is not evaluated, the function can test the head of the argument (which is Sin) and thus the first definition (associated with Sin[something]) applies. If however the value of the expression is stored in another variable, then by the time the pattern-matching takes place, there is no way for the function to test the head of an expression Sin[b] - all it has is a symbol <a> (again because <a> this time is held unevaluated) . This behavior may lead to rather subtle bugs in user-defined functions which use Hold attributes. One way out in this case would be to redefine the function as follows:

```
ClearAll[f];
f[x_] /; Head[Evaluate[x]] === Sin := x^2;
f[x_] := "Not sine";
SetAttributes[f, HoldFirst];
```

Here, by using Evaluate, we override the Hold attribute in that particular place and instruct the argument inside Head command to be evaluated. Now :

```
Clear[a, b];
a = Sin[b];
{a, f[a]}
```

```
{Sin[b], Sin[b]^2}
```

The case with a Sin[Pi] is lost however :

```
f[Sin[Pi]]
```

```
Not sine
```

If we think of it, this is still a more logical behavior, since it is more logical (or should I say more robust) to test the head of fully evaluated expression than the one which will evaluate to something else. If one wants to catch both cases (something that was Sin[expr] or something that will become Sin[expr]), this is also possible:

```
ClearAll[f];
f[x_] /; Head[Evaluate[x]] === Sin := x^2;
f[x_Sin] := x^2;
f[x_] := "Not sine";
SetAttributes[f, HoldFirst];
```

```
{f[a], f[Sin[Pi]]}
```

```
{Sin[b]^2, 0}
```

One may ask when in practice do such situations occur. More often than one may think, in fact. As a simple example, an expression may be assigned to a local variable in one function, which then passes this variable (with the "pass-by-reference" semantics) to another function which is supposed to both do a type-check and subsequently modify this variable. Such cases are relatively rare just because pass-by-reference semantics and in-place modifications are rarely used in "usual" *Mathematica* programming, but once you choose to program in this style (which occasionally is a good option), these sorts of problems will pop up much more often.

■ 4.9.7.3 Hold attributes and built-in functions

Many built - in commands have Hold attributes. For instance, the Set command has a HoldFirst attribute, since otherwise its l.h.s. would evaluate before Set will have a chance to assign anything to it (in case when the variable in the l.h.s. has a global value). SetDelayed has attribute HoldAll, since it does not evaluate also the r.h.s. of an assignment. Constructs such as Module, Block and With also have the Hold-All attribute, since they have to hold the code they enclose unevaluated until the naming conflicts are resolved. We could go on with this list, but let us just say once again that these attributes are very important.

■ 4.9.8 Attributes and the evaluation process

As we have discussed before, the evaluation process can be roughly represented by a repeated application of all available global rules to an expression and all of its parts, until the result no longer changes. We also mentioned that this is a very oversimplified picture. Now we can at least outline some other ingredients which make the evaluation process more complex.

One of such ingredients is the existence of attributes. You can not assign attributes in the form of local rules - they are essentially global properties of symbols. The presence or absence of attributes for a given symbol affects the way the expression involving this symbol is evaluated.

Another ingredient is the interplay of standard and non - standard evaluation. This is partly related to attributes through Hold attributes, but there are other ways to switch between standard and non - standard evaluation, such as using commands like Evaluate, Unevaluated, Hold, HoldPattern, etc.

Yet another distinction is that there are many more types of global rules than there are local ones. While local rules are basically either immediate (Rule) or delayed (RuleDelayed), global rules are additionally categorized by being OwnValues, DownValues, SubValues, UpValues, NValues or FormatValues (the latter three we did not have a chance to discuss yet). The category to which the global rule belongs, determines the way and order in which it is applied.

So, while the evaluation process generally is the repeated rule application, we can now see a bit better more of the ingredients that make it different and perhaps somewhat more complex than just a repeated application of all global rules.

■ 4.10 Advanced topic: parameter passing and local variables

In this section we will have a brief discussion on the interplay of parameter - passing and localization of variables with scoping constructs Module, Block and With and Function, which we promised in the section on the parameter passing (4.4.7).

It turns out that the situation is very similar for all these constructs, so we will discuss the Module case only. The main question is what happens if the name of some of the formal parameters coincides with a name of one of the local variables. Let me say straight away that this is a really bad practice which should be avoided since it brings nothing except bugs into the programs. Let us consider a simple example:

```
Clear[fM, a];
a = 5;
fM[x_] := Module[{x = 10}, Print[x]];
```

Here we set up a function < fM > with conflicting names of the parameter and a local variable, and just a global variable < a > assigned some value. Now we try a couple of inputs :

```
fM[5]
```

```
Module::lvset : Local variable specification {5 = 10} contains
    5 = 10, which is an assignment to 5; only assignments to symbols are allowed. ≫
```

```
Module[{5 = 10}, Print[5]]
```

We see what happened: the value for a formal parameter < x > (5 in this case) was textually substituted in all places where the literal < x > appears on the r.h.s., before any other evaluation (and name conflict resolution in Module in particular) took place. This is in full agreement with the general parameter - passing mechanism described earlier (section 4.4.7). But then, by the time Module actually started executing, we see what was inside - in particular, instead of the local variable initialization, we had in the variable declaration block a statement 5 = 10, which triggered an error message and resulted in Module returning unevaluated.

Conclusion: **it is an error to make a name of a local variable coincide with the name of any of the function parameters**.

We now try to call our function on a variable rather than a raw expression :

```
fM[a]
```

```
Module::lvset : Local variable specification {5 = 10} contains
    5 = 10, which is an assignment to 5; only assignments to symbols are allowed. ≫
```

```
Module[{5 = 10}, Print[5]]
```

The results are identical, because < a > evaluated to < 5 > before the function was essentially called (recall the standard evaluation mechanism).

Next, let us see what happens when a function has a Hold attribute for the parameter in question. We modify our code accordingly :

```
Clear[fMHold];
Attributes[fMHold] = {HoldAll};
fMHold[x_] :=
  Module[{x = 10}, Print[x, "   ", a, "  ", Unevaluated[a]]];
```

Here, we have included additional objects to be printed - in a second you’ll see why. Now let us test :

```
fMHold[5]
```

```
Module::lvset : Local variable specification {5 = 10} contains
    5 = 10, which is an assignment to 5; only assignments to symbols are allowed. ≫
```

```
Module[{5 = 10}, Print[5,    , a,   , Unevaluated[a]]]
```

The result here is essentially the same as before, because < 5 > is a raw object. Now let us see what happens if we call our function on a variable :

```
fMHold[a]
```

```
10   10  a$165
```

This output is quite interesting. The last output gives us a name that was internally associated with <a> in our code inside Module. It tells us that in this case, the local variable was initialized, and has shadowed the global parameter being passed. It is instructive to see exactly how this happened:

Step 1 : The symbol <a> in unevaluated form (due to a Hold attribute) is textually substituted everywhere where <x> stands inside the Module (r.h.s. of the function definition). At this point we have the code:

```
Module[{a = 10}, Print[a, "   ", a, Unevaluated[a]]]
```

Step 2 : A local variable < a > with a special name is initialized, and all occurrences of the symbol < a > in the code of Module are then associated with this local variable - just as if we had entered the above code from the keyboard.

Step 3: It is only at this point that the function would try to evaluate the passed parameter (since it was held unevaluated so far), but by this time all occurrences of <a> already correspond to the initialized local variable, which thus completely shadows the passed parameter value.

Step 4: The code is executed in the above form, with the results we just saw.

The conclusion is that if a given parameter is held by the function and if the passed object happened to be a global symbol with the head Symbol, then the parameter being passed is shadowed by a local variable.

This behavior looks more mild than the one before, but in fact it is worse. Because really, colliding names in this fashion is a bad mistake in both cases, but here it may go unnoticed, since it does not result in an explicit error.

If the passed held parameter is a composite expression, Module will at least generate an error message and return unevaluated, since it is illegal to name local variables in such way) .

```
fMHold[a[b]]
```

```
Module::lvset : Local variable specification {a[b] = 10} contains
    a[b] = 10, which is an assignment to a[b]; only assignments to symbols are allowed. ≫
```

```
Module[{a[b] = 10}, Print[a[b],    , a,   , Unevaluated[a]]]
```

The ***final conclusions*** are these :

1. There is no mystery in what happens in parameter and local variable name collisions - all the outcomes can be easily explained by the core parameter - passing mechanism based on textual substitution.
2. **It is always an error** to collide the names like this, but there are cases when this error may go unnoticed, and the parameter value be shadowed by a local variable.

A good news is that in version 6 such name collisions are usually detected and highlighted in red by the front - end.

The final comment here: this situation is not too specific to *Mathematica*. In C, for instance, it is also an error to name a local variable after one of the function formal parameters, and will result in an undefined behavior (at least, here it is not undefined). It is a different matter that the passed parameters themselves may serve in C as local variables, unlike in *Mathematica* (see a discussion in 4.4.7).

4.11 Pure functions

The notion of a pure function comes from the λ - calculus, and is widely used in functional programming languages, *Mathematica* in particular. From the practical viewpoint, the idea is that often we need some intermediate functions which we have to use just once, and we don' t want to give them separate names. Pure functions allow to use them without assigning them names, storing them in the global rule base etc. Another application of them is that while they can be assigned to some symbols, they exist independently of their arguments and can be called just by name with the arguments being supplied separately, so that the "assembly" to the working function happens already at the place where the function is used. Finally, these functions may be dynamically changed and modified during the program' s execution.

In *Mathematica*, the pure function can be defined in two (in principle, equivalent modulo some subtleties which we will discuss) ways: through the built-in function <Function> and through the so-called #-& notation (anonymous pure functions).

4.11.1 The # - & notation

We will first discuss the latter method. The idea behind it is to allow one to create functions which have no names and also no named arguments - completely anonymous pure functions. In this notation, the parameters of the function are denoted by sharp (#) plus the parameter index, like #1, #2, etc. If there is a single parameter, the index can be suppressed and we can just use #. The function ends with an ampersand <&>. It is recommended although not always required to put the entire function definition in parentheses (including the &), to avoid precedence- related bugs. This is because the ampersand & has a very low precedence and often a larger piece of code is interpreted as a part of the function definition, than meant by developer. This leads to bugs. One typical case where parenthesizing is absolutely necessary is when we provide a pure function as a sameness test in the SameTest option for the Union, Intersection or Complement commands.

Let us give some examples of pure functions.

4.11.1.1 Example: the squaring pure function

```
{(#^2 &)[1], (#^2 &)[2], (#^2 &)[Pi], (#^2 &)[10]}
```

```
{1, 4, π^2, 100}
```

The same can be done as follows :

```
ClearAll[f];
f = (#^2 &);
```

```
{f[1], f[2], f[Pi], f[10]}
```

```
{1, 4, π^2, 100}
```

To understand, how a given pure function expressed in such a notation will work, one has to get used to it a little. In the beginning it often helps to substitute the real values of the arguments for parameters #.

Let us note that the internal representation of < f > is different from the one we had for functions which were defined with the help of patterns. In particular, there are no DownValues associated with f (which means, no rules associated with the form f[something]) :

```
DownValues[f]
```

```
{}
```

Also, with this form of function definition, we can not associate several different definitions with the symbol f :

```
?f
```

```
Global`f
```

```
f = #1^2 &
```

We give a new definition :

```
f = #^4 &;
?f
```

```
Global`f
```

```
f = #1^4 &
```

This was possible with patterns, because f[pattern1] and f[pattern2] are different symbols. In our present case however, the best way to think about it is to think that the variable < f > received some value, which turned out to be not a number or another variable, but a pure function. Such a "variable" interpretation is also consistent with the fact that the rule for <f> is stored in OwnValues rather than DownValues.

- 4.11.1.2 Example: function which takes the first element from the list

```
Clear[takeFirst];
takeFirst = #[[1]] &;
```

We check :

```
takeFirst[Range[10]]
1
```

Pure functions defined in the # - & notation usually work faster than the pattern - defined ones (there is no pattern - matching going on in this case), but they require more care. In particular, it is less easy to organize the argument checks for them (although also possible, of course). Let us, for example, apply our function to a number instead of a list :

```
takeFirst[1]


Part::partd : Part specification 1⟦1⟧ is longer than depth of object. ≫
1⟦1⟧
```

To add the argument check, we would have to write something like this :

```
takeFirst = If[AtomQ[#], #, #[[1]]] &;
```

That is, to return the argument itself if it is an atom, and its first element if it is not:

```
{takeFirst[5], takeFirst[{3, 4, 5, 6}]}
{5, 3}
```

With the help of patterns, we could do the same as follows :

```
takeFirstP[x_ /; Not[AtomQ[x]]] := x[[1]];


{takeFirstP[5], takeFirstP[{3, 4, 5, 6}]}
{takeFirstP[5], 3}
```

Notice the difference in execution: when the pattern does not match, the function simply returns unevaluated. This allows to use the trick with the "soft" generation of error messages by adding a general catchall rule to catch all wrong arguments (see section 4.6.2.3 for an example). On the other hand, for pure functions, all cases have to be explicitly taken into account - in this sense they are more "rigid" entities.

Returning back to the previous example of the takeFirst function, we may indeed want it to remain unevaluated for atomic objects, rather than returning them. In this case, we really need a pattern - defined function.

■ 4.11.1.3 Example: a function of two variables

This function returns an intersection of two sets:

```
Intersection[#1, #2] &
```

```
#1 ⋂ #2 &
```

Let us use it:

```
Intersection[#1, #2] &[{1, 2, 3, 4}, {3, 4, 5, 6}]
```

```
{3, 4}
```

■ 4.11.1.4 Example: building a matrix of rank 1 from two vectors

This function creates a matrix of rank 1 for the two given vectors :

```
Outer[Times, #1, #2] &
```

For example :

```
Clear[a, b, c, a1, b1, c1];
Outer[Times, #1, #2] &[{a, b, c}, {a1, b1, c1}]
{{a a1, a b1, a c1}, {a1 b, b b1, b c1}, {a1 c, b1 c, c c1}}
```

■ 4.11.1.5 Example: supplying a list with its length

This function supplies a list with its length :

```
{#, Length[#]} &
```

For example :

```
{#, Length[#]} &[{5, 6, 7, 8, 9, 10}]
{{5, 6, 7, 8, 9, 10}, 6}
```

In all these examples, the same results would be obtained if the real values of the arguments were substituted, as is easy to check.

■ 4.11.1.6 If we supply a wrong number of arguments

If the pure function is given more arguments than it needs, extra arguments are ignored :

```
#[[1]] &[{1, 2}, {3, 4}]
1
```

This property has both advantages and disadvantages. The advantage is that if some other (higher - order) built - in function takes a given pure function as an argument but supplies more arguments than needed, I don' t need to modify my code (one such example is NestWhile when we need an access to more than just the most recent result - see chapter V).

The disadvantage is that one has to be extra careful. If the arguments were not supposed to be ignored, but the pure function is erroneous, such error is hard to catch. For comparison, for a pattern - defined function with the fixed number of arguments, for the wrong number of arguments the pattern simply won't match and the function will evaluate to itself (or, if one uses the trick with the catch-all pattern and error messages, the error message will be generated) - this situation is much better for debugging.

If, on the other hand, we supply less arguments than expected by the pure function, the error message will be generated :

```
(#1 + #2) &[1]
Function::slotn : Slot number 2 in #1 + #2 & cannot be filled from (#1 + #2 &)[1]. ≫
1 + #2
```

- 4.11.1.7 The Head and the FullForm of the pure function in #-& notation

The # - & form of a pure function is really just a convenient notation. The fundamental built - in function (head) used to create pure functions is always Function, as can be seen easily :

```
Head[#^2 &]
Function
```

This is how this pure function looks internally :

```
FullForm[#^2 &]
Function[Power[Slot[1], 2]]
```

- 4.11.1.8 SlotSequence and functions with variable number of arguments

It is sometimes needed to define a function of many variables, where their number is either not fixed or when the variables are not needed to be referred to separately. In this case, one can use the SlotSequence, which has an abbreviation ##. As an example, we may define our own Plus function as a pure function in the following fashion :

```
Clear[ourPlus];
ourPlus = Plus[##] &;
```

We can check :

```
{ourPlus[1], ourPlus[1, 2], ourPlus[1, 2, 3]}
{1, 3, 6}
```

It is less obvious, but one can also use SlotSequence even when one needs an access to individual variables. For example, we need a function which multiples its first argument by the sum of all the other ones, as a pure function. This does not seem possible to do since ## gives all arguments but does not directly allow to access the individual ones. This is not so however :

```
firstTimesSumRest = ({##}[[1]] * Total[Drop[{##}, 1]] &);
```

```
{firstTimesSumRest[1], firstTimesSumRest[1, 2],
 firstTimesSumRest[1, 2, 3], firstTimesSumRest[2, 3, 4, 5]}
```

```
{0, 2, 5, 24}
```

This shows how to access individual variables with SlotSequence - place ## in a list and then index the list. I deliberately ignored the variation of the SlotSequence which allows to supply first <n> arguments separately and the rest with SlotSequence, to illustrate the general way of accessing individual variables. With this variation, our function would be rewritten much more compactly as

```
firstTimesSumRest1 = (#1 * Plus[##2] &);
```

With the same results of course :

```
{firstTimesSumRest1[1], firstTimesSumRest1[1, 2],
 firstTimesSumRest1[1, 2, 3], firstTimesSumRest1[2, 3, 4, 5]}
```

```
{0, 2, 5, 24}
```

This form of the function is certainly shorter, but less general since I will not be able to access say the third argument without the use of the trick with the list indexing shown above.

One important limitation of the # - & notation for the pure functions is that it is not possible to assign attributes to them in this notation (unless some undocumented features are used). The advantage however is that functions in this form are typically faster than in other forms (patter-defined or defined through the Function construct with named arguments).

■ 4.11.1.9 A comment on function names

You may have noticed that I sometimes store the pure function definition in some variable. However, this is done purely for convenience in our examples, where I use these functions on several arguments, but don' t want to use any of the functional programming constructs which automate function application. When we come to that point, you will see that we will never need names for pure functions.

■ 4.11.1.10 Nesting pure functions in #-& notation

It often happens that inside one pure function there is another one, supplied to it as one of its arguments. The question is then whether or not we face any difficulties or ambiguities due to the same abbreviation for the function variables. As an example, consider a pure function which sorts its argument (list of lists is assumed), in an ascending order in the first elements of the sublists. This is how it will look like in the # - & notation :

```
Clear[sortFirstElem];
sortFirstElem = Sort[#1, First[#1] ≤ First[#2] &] &;
```

We see that the pure functions are nested one within another, and that there are two instances of #1 variable which have different meaning and, indeed, refer to different variables. Is it legal? The answer is yes, as long as one pure function is entirely contained in another one (more precisely, if there are no expressions such that for their evaluation, the variables of the nested functions have to be used together, simultaneously). Let us test it :

```
sortFirstElem[{{2, 3}, {1, 4}, {5, 7}, {3, 8}}]
{{1, 4}, {2, 3}, {3, 8}, {5, 7}}
Clear[sortFirstElem];
```

■ 4.11.1.11 Pure functions with zero arguments

This seems like a really weird construct, but it is in fact quite useful. Such functions are required basically when we want to supply some number or expression to a function as one of the arguments, while instead a function argument is expected. Then we need to "convert" our expression into an "idle" function, which will simply produce this expression regardless if its argument.

As a simple example, say we need to produce a list of ones, of the length 10 : {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}. This can be done by Table, of course, but alternatively by an Array command (which is somewhat faster). However, Array requires a function to be supplied, so the input like this:

```
Array[1, {10}]
{1[1], 1[2], 1[3], 1[4], 1[5], 1[6], 1[7], 1[8], 1[9], 1[10]}
```

does not work, as we can see. The solution is to "convert" the number < 1 > into a pure function with zero arguments. This is extremely easy to do - just add an ampersand to the end :

```
Array[1 &, {10}]
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

So, to summarize: adding an ampersand to the end of some expression (not involving anonymous variables - # symbols), converts this expression into a pure function with zero arguments, which is quite handy at times.

■ 4.11.1.12 Currying (partial application) with pure functions in # - & notation

Currying means the following: given a function of < n > arguments and passing to it the smaller number of arguments < k > (k < n, and argument positions not necessarily consecutive), we insert these arguments into their "argument slots" and create a new function of the remaining variables at run - time. Sometimes this is also called partial application of the function. If, for instance, we have a function <f> of two variables <x> and <y> from the sets X,Y, producing some result <z> from the set Z, then *f*:X x Y -> Z, while what currying does is <*curried f*>: X-> (Y -> Z). Thus, while initial function maps a cartesian product of X and Y onto Z, takes two arguments and maps them to a result (expression), curried function maps X onto a space of mappings (functions) Y->Z. The result of application of curried function is then always another function.

This technique is not very useful in procedural programming, since there it is the programmer himself who makes all the function calls. In the functional style however, we allow some functions to manipulate other functions, call them, etc. In this paradigm, it is quite often that one function expects another function as one of its arguments, and then frequently such a function does not exist and has to be created at run-time.

In some languages such as Ocaml, all functions are curried automatically. There is no built - in support for currying in *Mathematica*, but the compactness of the # - & notation makes it almost mindless to implement in each particular case. For example, say we have a function of two variables, to which we pass the first argument and then need to create a resulting function of a single (second) argument:

```
ClearAll[f];
f[x_, y_] := Sin[x * y];
```

Say, the first argument is Pi. This is how we create such a function :

```
f[Pi, #] &
f[π, #1] &
```

For example :

```
f[Pi, #] &[1]
0
```

This technique trivially generalizes to more arguments. We will make real use of it in chapter V on functional programming.

4.11.2 Pure functions defined with Function

Now, let us describe another way of defining pure functions - through the Function construct. It has the format: Function[{vars},body]. If there is a single variable, then the list braces are optional. Let us show how some of the previous examples would look in this notation:

The squaring function :

```
ClearAll[f];
f = Function[x, x^2];
```

```
{f[1], f[2], f[Pi], f[10]}
{1, 4, π^2, 100}
```

The function which take the first element :

```
Clear[takeFirst];
takeFirst = Function[x, If[AtomQ[x], x, First[x]]];
```

```
{takeFirst[a], takeFirst[{1, 2, 3}]}
{a, 1}
```

Intersection of two sets (lists) :

```
Clear[intSets];
intSets = Function[{x, y}, Intersection[x, y]];
```

```
{intSets[{1, 2, 3}, {2, 3, 4}], intSets[{1, 2, 3}, {4, 5, 6}]}
{{2, 3}, {}}
```

Matrix of rank 1, built out of 2 vectors :

```
Clear[extMultiply, a, b, c, d, e, f];
extMultiply = Function[{x, y}, Outer[Times, x, y]];
```

```
extMultiply[{a, b, c}, {d, e, f}]
{{a d, a e, a f}, {b d, b e, b f}, {c d, c e, c f}}
```

▪ 4.11.3 Differences between pure functions defined with Function and with # - & notation

It is important to note that there is no fundamental difference between functions defined with the # - & notation and functions defined with the Function command, in the sense that both definitions produce pure functions. There are however several technical differences that need to be mentioned.

The first one is that the Function[{vars},body] is a scoping construct, similar to Module, Block, With etc. This means in particular that the <vars> in the function body are localized to the body of the function, and have nothing to do with the global variables with same names, so we don't need to worry about name conflicts when defining pure functions with Function. This also means that, should we wish to nest the Function constructs one inside another, the possible name conflicts will automatically be resolved by the system (but I would recommend to read the corresponding sections of *Mathematica* Book and *Mathematica* Help to see precisely how they are resolved). Also, with Function we may nest functions in more general way than with #-& notation, which is illustrated on the following example:

▪ 4.11.3.1 Example: currying

This is a function which takes an argument and produces another function. That function takes another argument and returns the sum of these two arguments

```
Clear[nestedF];
nestedF = Function[x, Function[y, x + y]]
```

```
Function[x, Function[y, x + y]]
```

For example, we may "forge" a function which adds 3 to its argument, like this :

```
add3 = nestedF[3]
Function[y$, 3 + y$]
```

```
{add3[1], add3[5], add3[7]}
{4, 8, 10}
```

This is somewhat similar to a technique called currying in some languages. As this example illustrates, currying can be easily implemented in *Mathematica* through nested Function constructs, even though

Mathematica does not support it directly.

We can of course use our function directly on the 2 arguments

```
nestedF[3][1]
4
```

This should not be considered a function of two arguments however, since the first number defines the function which then takes the second number as a single argument. In particular, the evaluation of this function will be different from the evaluation of the more standard function of two arguments that we described before.

Returning to our original question of comparison of the # - & style and the style with Function, it is not possible (to my knowledge) to implement the above functionality with the # - & style, since here we have nested pure functions with one not entirely contained in the other one (in the sense described above - we needed the variables of both internal and external function simultaneously to do the computation). Thus, defining a pure function with Function is more general in this sense.

■

The other limitation of # - & approach, which we mentioned already, is that attributes can not be assigned to a pure function defined in this way (well, at least if one does not use undocumented features - see the section Attributes of Pure Functions in the Maeder's book). This is not the case with Function : it takes an attribute or a list of attributes as an optional third argument, which is a powerful capability.

■ 4.11.3.2 Example: an accumulator problem

To illustrate it, we will consider a model problem which Paul Graham used to argue in favor of functional languages (LISP).[13] : write a function, which takes an (integer) number <n>, and returns another function that takes any number and increments it by <n>. He emphasized 2 things: 1. returns a function, 2. this function not simply adds n to its argument, but increments it by n (that is, produces a side effects and changes a global value of the variable passed to it). Here is our solution:

```
Clear[incrementByN];
incrementByN[n_Integer] := Function[x, x += n, HoldFirst];
```

The conciseness of this solution arguably rivals the one in LISP. This was possible only because we could specify the HoldFirst attribute, which will actually allow the increment to be performed on the original variable rather than on what it would evaluate to. Let us now test it :

```
Clear[a, inc5];
a = 10;
inc5 = incrementByN[5];
inc5[a];
a
15
```

4.11.3.3 Example: the Listable SubValues hack revisited

In section 4.9.1.3, we considered a "hack" which solves the listability problem for a function in which not all list arguments have to be threaded upon. It involved introduction of an auxiliary function defined through SubValues. Here we consider an alternative (equivalent) implementation through the pure functions, which will be more compact.

I remind that the problem was for example to get the following evaluation : f[{1, 2}, {3, 4, 5}] -> {f[1, {3, 4, 5}], f[2, {3, 4, 5}]}. If we just give < f > Listable attribute, this won’ t work :

```
ClearAll[f];
SetAttributes[f, Listable];
f[{1, 2}, {3, 4, 5}]
```

Thread::tdlen : Objects of unequal length in f[{1, 2}, {3, 4, 5}] cannot be combined. ≫

```
f[{1, 2}, {3, 4, 5}]
```

The code below does the trick.

```
Clear[halfListable];
halfListable[f_, x_, y_] := Function[t, f[t, y], {Listable}][x]
```

Check :

```
ClearAll[f];
halfListable[f, {1, 2}, {3, 4, 5}]
```

```
{f[1, {3, 4, 5}], f[2, {3, 4, 5}]}
```

What happens is that the parameter < y > (on which the function does not have to be Listable), is textually substituted (recall parameter passing) into a pure function for which Listable attribute is given, at the moment of the construction of this pure function. Then, the constructed pure function is computed at argument < x > .

You can get even fancier and write a function which takes your given function name, but no arguments (x, y), and creates a pure function with < f > being embedded, and with the above behavior.

```
Clear[makeHalfListable];
makeHalfListable[f_] :=
 Function[{x, y}, Function[t, f[t, y], {Listable}][x]]
```

The advantage of this solution is that no specific arguments are involved whatsoever: you create a brand new function from < f >, with the functionality you want, only once, and then can use it many times later.

```
newf = makeHalfListable[f];
```

```
newf[{1, 2}, {3, 4, 5}]
```

```
{f[1, {3, 4, 5}], f[2, {3, 4, 5}]}
```

```
newf[{1, 2, 3}, {4, 5}]
```

```
{f[1, {4, 5}], f[2, {4, 5}], f[3, {4, 5}]}
```

The possible disadvantage is the overhead induced by extra < Function > being a scoping construct. However, this usually is a minor one.

■

To rehabilitate the # - & notation, the functions defined with it are usually faster than those defined with the Function construct (my guess is that this has to do with the scoping overhead - Function with named variables is a scoping construct). They are also more compact.

The functions defined in the # - & notation can be nested with those defined with Function. We will see several examples of such mixed constructs later.

To conclude our section on pure functions, let me emphasize once again that there is no natural mechanism for them to perform arguments checks, such as (restricted) patterns in the pattern - defined functions. Pure functions really play a different role and are used in different settings. One can get the most out of the pure functions when they are used within a functional programming style, very often as arguments of other (higher-order) functions. When we come to the next chapter which describes it, their usefulness will become much more apparent.

4.12 Functions with defaults and options

It is often needed that part of the arguments are optional in the sense that some of the "argument slots" may be either used or not, and the function has to do meaningful things in both cases. In *Mathematica*, like in many other modern languages (Python comes to mind), there are two mechanisms to provide this functionality: default values (for positional arguments), and Options (this corresponds to "named arguments").

What is perhaps unusual and specific to *Mathematica* is that neither of these mechanisms require some new special syntax in the sense that it has to be added externally into the system. Rather, both of them exploit some features already present in the system, such as optional patterns (see section 4.2.9) or non - commutativity of the rule substitutions (see section 4.2.2.).

■ 4.12.1 Functions with defaults

Default arguments are those which we can leave out when calling a function, in which case there are some default values that the function will use for these arguments. The matching between missed arguments and values is based on the positions of the arguments in this case. In *Mathematica*, this mechanism is realized through optional patterns (section 4.2.9). We will give just a few simple examples of such functions

Here we define a function which sums all its arguments, and has the last two arguments optional, with default values being 1 and 2 :

```
ClearAll[f];
f[x_, y_: 1, z_: 2] := x + y + z
```

Check :

```
{f[1], f[1, 3], f[1, 3, 5]}
{4, 6, 9}
```

The default patterns may be interspersed with the patterns for fixed arguments. The rule how the arguments are filled in is more complicated in this case: the pattern - matcher first determines how many optional arguments can be filled (from left to right), and then fills all the arguments from left to right, fixed and optional at the same time (not like fixed first, optional next). This is the consequence of the general way of how the pattern - matcher works, but one conclusion is that it is best to move all optional (default) arguments at the end of the argument list, to have a better idea of the order in which the arguments are filled. Here is an illustration :

```
Clear[g];
g[x_, y_: 1, z_, t_: 2] := {x, y, z, t}
```

Check :

```
{g[a], g[a, b], g[a, b, c], g[a, b, c, d]}
{g[a], {a, 1, b, 2}, {a, b, c, 2}, {a, b, c, d}}
```

That' s about all I will say for default arguments. The more complete treatment can be found elsewhere [6,7,9].

■ 4.12.2 Functions with options

Options are a mechanism to implement named arguments in *Mathematica*. They are especially convenient for the "end" functions which interface with the user or programmer. A typical example of use of options is to manipulate the format of data output on the screen, or to indicate the name of the particular method or algorithm to be used in some numerical computation.

The options mechanism is an example of use of the non - commutativity of rules application. We will illustrate this in detail on one particular model example.

■ 4.12.2.1 Example : selecting and printing prime numbers

The problem and the solution

Our task here will be to select and print prime numbers contained in a given list of numbers. The option will be responsible for the number of primes which have to be printed. But first, let us write a function which will display all the numbers found:

```
Clear[findPrimes];
findPrimes[x_List] := Module[{res},
   res = Cases[x, _ ? PrimeQ];
   Print[res];
   res];
```

Note that this function not only returns a list of all the numbers found, but also prints this list.

```
findPrimes[Range[50]]
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}
```

We can block the output with the semicolon, but the printing will of course still happen :

```
findPrimes[Range[50]];
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}
```

Now, we want to include an option to print no more than the first < n > numbers, and say the default option will be < n > = 10. This is how the code will look like :

```
Clear[findPrimes, DisplayN];
findPrimes[x_List, opts___ ? OptionQ] :=
  Module[{res, printnumber, printres},
   printnumber = DisplayN /. Flatten[{opts}] /. DisplayN → 10;
   res = Cases[x, _ ? PrimeQ];
   printres =
    If[Length[res] ≤ printnumber, res, Take[res, printnumber]];
   Print[printres];
   res];
```

Let us first check that it works correctly, and then dissect the code to understand how it works. First, we will not explicitly use an option:

```
findPrimes[Range[50]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}
```

We see that 10 numbers were printed. Note that the result of the function execution is still a list of all primes found - our option only affects the printing. Let us now explicitly define an option - say we want to output only the first 5 numbers :

```
findPrimes[Range[50], DisplayN → 5];
```

```
{2, 3, 5, 7, 11}
```

Code dissection

Now, let us dissect the code. The main line which is at the heart of the options mechanism, is this one : **printnumber = DisplayN /. Flatten[{opts}] /. DisplayN → 10;** What happens here is the following:

1. The name of the option itself - <DisplayN> - does not and should not have any value. It is important to note here that all options are defined as some rule. In this case, the rule is: everywhere where the literal <DisplayN> is encountered, replace it by some number, say 10 (or 5 or whatever).

2. The variable <opts> in the pattern is a pattern tag with the BlankNullSequence (triple underscore), which means that the entire function pattern will work even if nothing will be entered for the second argument - < opts > . This is why we call it options, and our function worked in the first of the two cases above.

3. A built - in predicate **OptionQ** checks that < opts > is a rule or a list of rules, and not something else. In particular, the following input will not evaluate :

```
findPrimes[Range[50], 5]
findPrimes[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
  19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
  36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50}, 5]
```

4. The operation Flatten[{opts}] is needed because in more complicated cases options may represent nested lists of rules. But the rule replacement command ReplaceAll (/.) works the way we want it here only with simple (not nested) lists of rules. Thus, an additional list structure, if present, has to be destroyed. Flatten is used to do this.

5. Now we come to the essence: suppose that among the options passed to our function through the <opts> argument, is an option related to DisplayN literal. For example, in our case we used DisplayN->5. Therefore, Flatten[{opts}] gives {DisplayN->5}. Then, in the expression DisplayN /. Flatten[{opts}] the rule will apply, and as a result, this expression will be replaced by 5. Then, since the rules are applied from the left to the right (left-associatively), the last rule in the chain will look like 5/.DisplayN->10. But since the literal <DisplayN> is no longer present in our transformed object (5), the rule will not apply and then the whole expression DisplayN /. Flatten[{opts}]/.DisplayN->10 will be equal to 5, and this number will be assigned to the variable <printnumber>, which really controls the number of printed primes.

If, however, there will be no option with the DisplayN literal, then the literal DisplayN will "reach" the last rule /.DisplayN->10 without change, so then this rule will apply and the variable <printnumber> will receive the value 10.

4.12.2.2 Option names

If you decide to set up some options for a function you are writing, it may be a good idea to Protect the associated option name. Because, as it is clear from the discussion above, the option mechanism will be immediately broken if the option name accidentally gets some value. In fact, the bugs associated with this are sometimes very hard to catch.

4.12.2.3 Passing options to other functions

If a function receives some options, and then calls another function which also uses options, then the calling one can pass the options it received (or some of them) to the called one - this transfer does not require any changes in the base mechanism of options. On the other hand, the possibility of options passing makes it possible to have a very flexible control over the program.

To illustrate option passing, let us reformulate our previous problem somewhat. We will now have two options : one tells whether or not to print the result - call it < printOption >, and another one will instruct how many primes to look for (previously we were collecting all) - call it <searchNumber>. Also, we will now package the search as a separate function, which we will call findPrimes, and the main (calling) function we will call showPrimes. The variable corresponding to the searchNumber option call <snumber>.

```
Clear[findPrimes, printOption, showPrimes];

 findPrimes[x_List, opts___?OptionQ] := Module[{snumber},
    snumber = searchNumber /. Flatten[{opts}] /. searchNumber → 10;
    Cases[x, _?PrimeQ, 1, snumber]];

 showPrimes[x_List, opts___?OptionQ] := Module[{res, printq},
    printq = printOption /. Flatten[{opts}] /. printOption → True;
    res = findPrimes[x, opts];
    If[printq, Print["This is the printed result:  ", res]];
    res];
```

Check now :

```
showPrimes[Range[50]]
```

```
This is the printed result:  {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

```
showPrimes[Range[50], searchNumber → 5]
```

```
This is the printed result:  {2, 3, 5, 7, 11}
```

```
{2, 3, 5, 7, 11}
```

```
showPrimes[Range[50], searchNumber → 5, printOption → False]
```

```
{2, 3, 5, 7, 11}
```

We see that our function became very flexible. The point is that using options, we can alter the execution of more than one function, and this happens automatically (when the program is written correctly). It is easy to see what happens when we pass options in a "cascading" way. Even if every option can only be True or False, we have the total number of possible scenarios equal to 2^(number of options). And because the options are passed to auxiliary functions, we relegate the corresponding decisions to be made inside those auxiliary functions rather than one big main function - dispatcher. Thus, we take this load off the main function, which leads to a better program design.

■ 4.12.2.4 Filtering options

To cleanly implement this procedure however, we need to ensure that each function only receives the options that it understands (well, in principle nothing bad happens when the option is not understood by a given function - it is then just ignored, since it does not trigger any rule, but it is considered a better programming style not to send foreign options to functions. Besides, the built-in functions will issue error messages and return unevaluated if some unknown options are passed to them). There is another mechanism called filtering options, which is designed to do just that. This mechanism is beyond the scope of our discussion, but is is described in many places [2,6,7,9]. Let me just mention that in the version 6 there are several new functions such as FilterRules which are specially designed to simplify filtering options.

4.12.2.5 Advanced topic: globally defined defaults for options

As an alternative to explicitly indicating all default option values in the function itself ("hard-coding" them), options for a function can be defined globally, with the help of Options built - in function. For example, here we define an option Heads -> True, plus some other option, for some symbol (function) < f > :

```
ClearAll[f];
Options[f] = {Heads → True, anotherOption → someValue}
{Heads → True, anotherOption → someValue}
```

The Options command can also be used to monitor which options are known to the system for a given function, and their default values. Basically, Options[function] is a container to keep function' s options and their default settings:

```
Options[f]
{Heads → True, anotherOption → someValue}
```

If one needs to modify the default value of some of the known (to the system) options of a given function, one can use SetOptions command. This way it is not necessary to retype all the options which do not change:

```
SetOptions[f, anotherOption → differentValue];
Options[f]
{Heads → True, anotherOption → differentValue}
```

This however will not work if the option is not known to the system :

```
SetOptions[f, thirdOption → itsValue]
SetOptions::optnf : thirdOption is not a known option for f. ≫
SetOptions[f, thirdOption → itsValue]
```

The option has to be known to the system before SetOptions can be used with it.

For built - in functions, it is anyway a mistake to introduce unknown options, but for user - defined ones it makes perfect sense - at the end, we have to define the global defaults (if we decide to use them) at some point! When one decides to use the global defaults through Options (and this has advantages we will discuss in a moment), then normally one sets all the option defaults at once with a single statement Options[function] = {option1 -> default1, ..., optionk -> defaultk}. Adding new options at run - time is a bad idea in most cases, and also not possible if the function gets Protect-ed.

So, why this mechanism is any better than the one where defaults are "hard-coded" into a body of the function? Primarily, for a better code readability and maintenance. Usually, globally defined defaults are used when writing packages, and then all the defaults for all options for package functions are usually defined in the beginning of the package and can be easily inspected and changed later on.   Also, it is often convenient if the options of a given function (in their current state) have to be either inspected or passed to another function (possibly after having been filtered). It is hard to imagine how one could do it without this mechanism, given that the calling function which passes them may be not the one whose options are being passed.

The possible danger of this mechanism is that one may redefine the default values for function's options at some point in the program, and then this function used after that point will use the new defaults in all places where it is called. I would not recommend resetting function's options (especially for built - ins) globally if your program will be used by other people. It is always possible to just simply call the function of interest with needed option values passed to it explicitly, or, if it has to be called many times or you want to hide the implementation details, write a wrapper package where you can define your own function like this - this is safer.

This has also implications for writing packages: for all (especially built - in) functions used, always pass explicitly all the options they have with the values you need, even if these values are system defaults - the user of your package may have redefined the defaults before loading your package.

Returning to the semantics of options, the above mechanism converts the idiom <optionvar = OptionName /. Flatten[{opts} /. OptionName -> DefValue> to <optionvar = OptionName /. Flatten[{opts} /.Options[thisFunction]>. Because of this, if you still decide to change options globally, I would not recommend assignments such as Option[function] = {list of options} (as those described above), for the following reason: with this assignment (unlike when you use SetOptions), you have to be careful to list all the options with their current defaults, not just those that you are currently changing. But if you miss some, this may result in a "dangling" variable <optionvar> for the option(s) you miss: say you have a line of code *Module[{...,optionvar = ourOption /. Flatten[{opts} /.Options[thisFunction]}, body].* If you accidentally delete the rule for <ourOption> as a result of manipulations with Options[yourfunction], and pass no explicit value for this option through <opts> either, the <optionvar> variable will be initialized with a literal <ourOption>, rather than the value. Using SetOptions is much safer.

In fact, if the symbol of your function is protected (has a protected attribute), the system will automatically forbid assignments Options[function] = ... :

```
Clear[g];
Options[g] = {firstOption → value1, secondOption → value2};
Protect[g];
```

We try now :

```
Options[g]
{firstOption → value1, secondOption → value2}
Options[g] = {thirdOption → value3}
Set::write : Tag g in Options[g] is Protected. ≫
{thirdOption → value3}
Options[g]
{firstOption → value1, secondOption → value2}
```

If you went so far as to define global option defaults for your function, it probably then makes sense to Protect it, so that option changes will only be possible through the SetOptions route. I remind however that the Attributes of protected functions can still be modified.

As we have noted before, Clear will not clear options associated with the symbol:

```
Clear[f];
Options[f]
{Heads → True, anotherOption → differentValue}
```

You have to use ClearAll to remove option defaults :

```
ClearAll[f];
Options[f]
{}
```

To clear definitions associates with a Protected symbol, you have first to Unprotect it.

■

To summarize: functions with options enhance the flexibility and versatility of functions you are writing (and built-ins as well, of course).

```
Clear[findPrimes, showPrimes];
```

■ Summary

In this rather long chapter we have looked at rules, patterns and functions. From the practical viewpoint, and given that the most effective programming style in *Mathematica* is a functional one, we are more interested in functions. However, in *Mathematica* function definitions really are rules, and thus we have to understand how to deal with rules and patterns, in order to handle functions.

We have considered various types of patterns and rules. For patterns, we considered various building blocks, as well as mechanisms to construct restricted, or conditional, patterns. We have also described many built-in functions that take patterns as their arguments, such as Cases, Position, MemberQ, etc.

Then we saw many examples of functions of a single or multiple arguments, defined through patterns. We also saw that a function may have simultaneously many definitions, corresponding to different patterns. This is a very powerful capability, which allows to make the code both safer and easier to read.

Apart from the pattern - defined functions, we have considered another very important class of functions - anonymous, or pure functions. We discussed how to define and use such functions.

Functions may have some properties which affect the way they are evaluated. These properties are called attributes. We considered several important attributes : Listable, Flat, Orderless, Protected, and HoldFirst, HoldRest and HoldAll attributes and illustrated their use and effect.

In many cases the code for a function is longer than just a single line, and also some intermediate variables are needed to store temporary values. We discussed the scoping constructs that exist in *Mathematica* for localizing such variables - Module, Block and With.

When it is desired to substitute default values for some of the arguments based on the argument positions, one can implement functions with default values through the use of optional patterns. Alternatively, if a function has many parameters which determine its behavior and which are typically set to some default values, named arguments are preferred. This other possibility to provide the alternative values for these arguments is realized in the mechanism of options. We introduced options and illustrated their use on a simple example.

Now that we have understood both lists and functions, it is time to combine the two topics to get something really powerful: functional programming. This is a topic of the next chapter.

V. Functions on lists and functional programming

5.1 Introduction

Functional programming is a programming paradigm in which the central role is played by application of functions, to both data and other functions. Functions themselves are treated as data, and thus can be arguments of other functions. Since any data structure can be represented as a (possibly nested) list, functional programming then is about application of functions to lists.

There are important differences between functional programming in *Mathematica* and other functional languages (LISP). One difference is that recursion on lists is inefficient in *Mathematica* since lists are implemented here as arrays rather than linked lists. Another difference is induced by the rule - based nature of *Mathematica* in the way it reflects itself in function definitions (in the case of pattern - defined functions) and evaluation procedure (global rule base, expression rewriting etc).

Apart from being concise, the functional programming style is usually the most efficient in *Mathematica*. Also, although we do not consider it in this chapter, in *Mathematica* it is possible to use functional programming techniques on expressions more general than lists - basically, on general *Mathematica* expressions. This is a very powerful capability.

A few words about the role of this chapter. Perhaps, it will not be an overestimation to say that this is the most important chapter of all. Mainly, this is because it introduces certain new programming style and a number of programming idioms, which will be heavily used in all later chapters and which together form a different level of programming in *Mathematica*, not just as technical tricks, but as a new way of thinking about the problems. Those who are familiar with functional programming languages may find some of the material familiar. However even for them, there will be a lot of new information specific to *Mathematica*, which must be used in order to program it most efficiently.

Examples in this chapter play an important role in the overall exposition of the material. Many of them are used to illustrate some important concepts or subtleties, since I believe that any new idea is best understood when illustrated by a few examples. To get a complete grasp of this chapter, it is recommended to go through all examples, and pay attention to the annotations attached to them. Some of the examples use admittedly rather artificial settings. This is because their primary goal is to illustrate a given language idiom in a rather simple situation.

5.2 Core higher-order functions

5.2.1 Introduction

Roughly speaking, functional programming (FP) in *Mathematica* consists of application of functions to *Mathematica* normal expressions. A very important special case is when the normal expression is a list (i.e, it's Head is List), and we will be mostly concerned with this one in this chapter. However, most of what can be done with lists within functional programming paradigm, can also be done with general normal expressions.

Two things make FP non-trivial:

1. Functions can take other functions as their arguments (this has an analog of function pointers in C), but also can create and/or return new functions at run-time, be that pure functions or pattern-based ones. The latter capability has no direct analog in procedural languages, where the functions definitions are determined at compile time.

2. Lists can be made of arbitrary *Mathematica* expressions, be those atoms (numbers, strings or symbols), or normal expressions. In particular, one may consider nested lists which can be used to implement various data structures (lists, trees etc). This also means that a single list may contain objects of different types.

There are a few characteristic features of the functional programming style which I would like to mention here in order to give a flavor of it. One is that side effects (such as variable assignments) are (almost) absent. Another is that loops are very rarely used if at all. In *Mathematica*, this is not an absolute restriction however - it is just more natural to use other constructs, as we will see below.

Functions which take other functions as their arguments, are called higher order functions. On the conceptual level, there are just two most important built-in higher order functions in *Mathematica* - Map and Apply. On the practical level, these two are still most frequently used functions, but some "less fundamental" operations are still needed so often that special built-in functions exist for them, and are also quite handy.

We will now go through several most often used built-in higher order functions, illustrating their use with examples. Since they serve as building blocks of most of functional programs, one can do quite a lot being equipped with just these functions.

5.2.2 Map - simplest form

This is one of the two most fundamental built-in higher order functions, and by far the most frequently used one. Very roughly, one may say that it is used to replace loops within the FP paradigm.

In it's simplest form, Map takes two arguments: another function - let us call it <f> - of a single argument (I hasten to comment that the function may have no name, if it is a *pure function* (see section 4.11)), and an expression - let us call it <expr>, on which this function should by mapped. If <expr> is an atom, it is returned back. If <expr> is a list (or other normal expression), then f is applied to every element of the list, and the resulting list is returned.

■ 5.2.2.1 Simple examples of use

A few simple examples

```
Clear[f];
Map[f, a]
```

```
a
```

```
Map[f, {a, b, c}]
```

```
{f[a], f[b], f[c]}
```

In the above, the function <f> did not have any definition yet. Let us define it

```
f[x_] := x^2;
```

Now:

```
Map[f, {a, b, c}]
```

$\{a^2, b^2, c^2\}$

```
Map[f, a]
```

```
a
```

■ 5.2.2.2 Map is a replacement for a loop

Now we can see how it replaces a loop: in a procedural version, we will need something like this

```
Module[{i, len, expr, newexpr},
 For[i = 1; expr = {a, b, c};
  len = Length[expr]; newexpr  = Table[0, {len}],
  i ≤ len, i ++, newexpr[[i]] = f[expr[[i]]]];
 newexpr]
```

$\{a^2, b^2, c^2\}$

Notice that I deliberately packaged the code into a Module, to make variables like <i>, <expr>, etc local and avoid global side effects.

So, even here we can already see what we win by using Map:

1. We don’t need to introduce auxiliary variables
2. We don’t need to know the length of the list beforehand
3. The code is much more concise.
4. It is not obvious at all, but in most cases the code is faster or much faster.

What really happens is that a copy of the original list is created, and then all the operations are performed on the copy. The original list remains unchanged.

5.2.2.3 Using Map with a pure function

To see how to use a pure function inside Map, let us just reproduce the previous result:

```
Map[#^2 &, {a, b, c}]
```

```
{a^2, b^2, c^2}
```

In this case, there is no need for a function name. Also, pure functions (especially when used in Map) may be more efficient that the pattern-defined ones, because no pattern-matching is taking place. But this also means less protection from the bad input, as we discussed before.

■ 5.2.2.4 Shorthand notation and precedence

As for many common operations, there is a shorthand notation for Map - a symbol **/@** (slash - at). The usage is <**(function/@expression)**>. For example:

```
(f /@ {a, b, c})
```

```
{a^2, b^2, c^2}
```

```
(#^2 & /@ {a, b, c})
```

```
{a^2, b^2, c^2}
```

One may use either literal <Map>, or </@>. Their action is equivalent as long as one always keeps the parentheses as shown above. In many cases, like above, they can be omitted:

```
f /@ {a, b, c}
```

```
{a^2, b^2, c^2}
```

```
#^2 & /@ {a, b, c}
```

```
{a^2, b^2, c^2}
```

In general however they are needed to avoid precedence-related bugs. For instance, in the following example: we want to first map <f> on the list, and then square the result. For the latter part , we use a pure function in the prefix notation (function@expression). What we should get is {a^4,b^4,c^4}. Instead:

```
#^2 &@f /@ {a, b, c}
```

```
{f^2[a], f^2[b], f^2[c]}
```

What happens is that the function symbol is squared, and only then Mapped. Now:

```
#^2 &@(f /@ {a, b, c})
```

```
{a^4, b^4, c^4}
```

This sort of problem is impossible with the use of literal Map:

```
#^2 &@Map[f, {a, b, c}]
```

```
{a^4, b^4, c^4}
```

Also, literal Map often makes a program easier to read. So my advice would be to use it until you become experienced with it. In practice however, the </@> form is often more handy.

5.2.2.5 Associativity

Map operation is right-associative, which means that parentheses may be omitted in the following code:

```
g /@ g /@ {a, b, c}
{g[g[a]], g[g[b]], g[g[c]]}
f /@ f /@ {a, b, c}
```

$\{a^4, b^4, c^4\}$

5.2.2.6 More examples

Let us now consider a few of the more interesting examples.

Here, by mapping Range on a list produced by another Range, we create a following list of depth 2.

```
Map[Range, Range[4]]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}}
```

Or equivalently

```
Range /@ Range[4]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}}
```

This will take a list of lists and return a list of their first elements:

```
Map[First, {{a, b}, {c, d}, {e, f}, {g, h}}]
{a, c, e, g}
```

or

```
First /@ {{a, b}, {c, d}, {e, f}, {g, h}}
{a, c, e, g}
```

This will take a list of lists, and return a list of lists of all subsets of initial lists:

```
Map[Subsets, {{a, b, c}, {d, e}}]
{{{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}},
 {{}, {d}, {e}, {d, e}}}
```

You could have noticed already that all these examples essentially need a single loop which is replaced by Map. While by itself this is not a big deal, the main profit is another layer of abstraction - we no longer need variables and assignments, and thus don't need to worry about them. For example, a problem of checking the array bounds just does not exist in this approach, without any toll on performance (apart from that generally induced by *Mathematica*'s symbolic engine). Another added advantage is that the list of results is produced by Map internally, and thus efficiently, while in a procedural version we have to do it by hand, which is inefficient in *Mathematica* as we already discussed before.

5.2.2.7 Mapping a function of several arguments, with all arguments but one fixed

Let us consider another situation: what if we want to Map a function of more than one argument, but where all arguments but one are fixed. For example:

```
Clear[f, a];
f[x_, y_] := Sin[x + y];
```

And we want to Map it on a list {1,2,3,4,5}, with the variable <y> fixed at value <a>.

Perhaps the best solution to this problem is obtained by using the built-in <Thread> command. But here, for the sake of example, we will see how we can get it with Map. Later we will return to this example again and show a solution using Thread.

One way is to define an auxiliary function g, as follows:

```
Clear[g];
g[x_] := f[x, a];
```

Now we can use Map:

```
Map[g, Range[5]]


{Sin[1 + a], Sin[2 + a], Sin[3 + a], Sin[4 + a], Sin[5 + a]}
```

If we need to solve just this problem, the disadvantage of the present solution is that we have to introduce an auxiliary function, which we only need once. If, on the other hand, we want to perform this operation more than once, the disadvantage is that we make a function <g> implicitly depend on the global variable <a> - this is a recipe for disaster. The better solution would be to use a pure function:

```
Map[f[#, a] &, Range[5]]
{Sin[1 + a], Sin[2 + a], Sin[3 + a], Sin[4 + a], Sin[5 + a]}
```

We have essentially created a curried function along the lines of section 4.11.1.12. In this case, the function is constructed on the spot, and no name or global definition is associated with it. This may be considered as one of the idioms, which is good to remember. However, keep in mind that in many cases one can use another built-in function Thread (to be covered soon) which is specially designed for this sort of situations and may be faster.

```
Clear[f];
```

▪ 5.2.2.8 How to prevent Map from mapping on the entire list

I mentioned before that Map in its simplest form is a replacement for a loop. Since it is often useful in procedural approach to exit the loop abnormally with a Break[] command, let us discuss its analog for Map.

In the *Mathematica* model of computation, the best style of programming consists of operating on any composite object as a whole, and avoid breaking it into pieces, whenever possible. The functionality of Map is in full agreement with this principle - it Maps a given function on an entire list, whatever its length

In the *Mathematica* model of computation, the best style of programming consists of operating on any composite object as a whole, and avoid breaking it into pieces, whenever possible. The functionality of Map is in full agreement with this principle - it Maps a given function on an entire list, whatever its length is. If however we want it to stop abnormally, the only way to do it I am aware of is to throw an exception. However, ***if*** the result one is interested in is a resulting list (up to the point where the exception was thrown), then one has to either introduce auxiliary variables, or (much better), use a more sophisticated technique based on Reap-Sow operators (only version 5 onward).The Reap - Sow technique will be covered in detail in part II.

Consider an example: we want to map a function squaring its argument on a list of random numbers, but stop as soon as we encounter first non-positive number, and return the part of a list which has been processed.

Here is a list:

```
testlist = Table[Random[Integer, {-1, 10}], {15}]
{1, 2, 10, -1, 7, 9, 6, 5, 6, 4, 1, 0, 9, -1, 0}
```

This will be a solution

```
Module[{result = {}}, Catch[
  Map[If[# > 0, AppendTo[result, #^2], Throw[result]] &, testlist]]]
{1, 4, 100}
```

While this solution is not ideal in many ways, and uses some operations (Catch - Throw) not covered yet, my point here is to illustrate 2 things:

1. ***It is*** possible to prevent Map from going through an entire list even though Map is a built-in command without such explicit capability (it sweeps through an entire list by default).

2. There is a significant price to pay in doing so. Here, we paid by:
a. Introducing a variable <result> (and then the Module construct to make it local)
b. Making a function to be Mapped more complicated (now it contains an If statement).
c. <result> is continuously appended in place - this will become inefficient for large lists. For this last point, there are workarounds but they will make the code more complicated.
d. We loose the natural advantage that Map gives us: Map normally produces the resulting list for us. Here we instead create our own resulting list (<result> variable), and inevitably do it inefficiently as compared to a built-in Map. In fact, in this example the use of <Scan> command instead of Map would be more appropriate, but we have not covered it yet.
e. The code is less concise.

So, my suggestion would be to try designing a program such that this kind of interruption of Map is not needed - it is possible to do this in most cases. If there is no other way - then use Throw and Catch as above (avoid however appending large lists in place - there are better techniques of list creation to be discussed later) . In fact, in many cases it may be more efficient to first Map the function on an entire list, and then pick from that list only certain elements.

■ 5.2.2.9 Interaction with the procedural code

It is possible to enhance somewhat the functionality of Map by embedding some procedural code (essentially, side effects) inside the function being mapped. One can also view this as continuous run - time redefinitions of the function being mapped. Whatever the interpretation, *Mathematica* allows for such things, which is often handy. A few examples:

■ *5.2.2.9.1 Example: partial sums*

This will create a list of partial sums :

```
Module[{sum = 0}, sum += # & /@ Range[10]]
```

or, we can define a function

```
Clear[partSum];
partSum[x_List] := Module[{sum = 0}, sum += # & /@ x];
```

```
partSum[Range[15]]
```

```
{1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120}
```

■ *5.2.2.9.2 Example: simulating MapIndexed*

Here we will mimic the action of the MapIndexed function (covered next) in its simplest form (when it is used on a flat list) - it supplies the position of the element in a list as a second argument to the function being mapped on the list.

```
Module[{pos = 1}, f[#, {pos ++}] & /@ Range[10, 20]]
```

```
{f[10, {1}], f[11, {2}], f[12, {3}],
 f[13, {4}], f[14, {5}], f[15, {6}], f[16, {7}],
 f[17, {8}], f[18, {9}], f[19, {10}], f[20, {11}]}
```

We can again package this as a function :

```
Clear[myMapIndexed];
myMapIndexed[f_, x_List] :=
  Module[{pos = 1}, f[#, {pos ++}] & /@ x];
```

■ *5.2.2.9.3 Example: moving average revisited*

We can implement a version of moving average function by constantly updating a "running" list of neighbor points during mapping. Here we Map a function that does it, on a list of first 15 natural numbers, and average each number with 2 neighboring numbers on each side :

```
Module[{avlist = {}, result}, CompoundExpression[AppendTo[avlist, #];
    If[Length[avlist] ≥ 5, result = Total[avlist] / 5;
     avlist = Rest[avlist]; result]] & /@ Range[15]]
```

```
{Null, Null, Null, Null, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}
```

We have to delete the first 2 m results, since they will be Null. Here is the resulting function :

```
Clear[movAverage];
movAverage[x_List, m_Integer] /; Length[x] > 2 m :=
  Drop[Module[{avlist = {}, result},
    CompoundExpression[AppendTo[avlist, #]; If[
        Length[avlist] ≥  2 m + 1, result = Total[avlist] / (2 m + 1);
        avlist = Rest[avlist]; result]] & /@ x], 2 m];
```

Check :

```
Range[10]^2
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
movAverage[Range[10]^2, 1]
```

$$\left\{\frac{14}{3}, \frac{29}{3}, \frac{50}{3}, \frac{77}{3}, \frac{110}{3}, \frac{149}{3}, \frac{194}{3}, \frac{245}{3}\right\}$$

This is another implementation where the idea of using side effects is pushed to the extreme :

```
Clear[movAverageAlt];
movAverageAlt[x_List, m_Integer] /; Length[x] > 2 m :=
  Module[{n = 1},
   Total[Take[x, {n, 2 m + n ++}]] / (2 m + 1) & /@ Drop[x, 2 m]];
```

Notice that here, it does not use any information from the list on which the function is mapped - this is a pure function with zero arguments (see section 4.11.1.11), so we could just as well map on a list of zeros or ones - it just needs to have the right length. C programmers may worry about the line {n,2m+n++}, since its value depends on which part of the list is evaluated first. But here this is system-independent, since the standard evaluation procedure applied to a simple flat list prescribes that the parts of such list will always be evaluated from left to right.

We can do some efficiency tests

```
movAverage[Range[10 000], 10]; // Timing
{0.641, Null}
movAverageAlt[Range[10 000], 10]; // Timing
{0.34, Null}
```

We notice that the second version is about twice faster. Let me mention that these implementations are not very efficient, and the purpose of this example was to demonstrate the construction used here.

This is one of the most efficient implementations, which we covered before in section 3.8.1.2, for comparison :

```
Clear[movingAverage];
movingAverage[x_List, m_Integer] :=
  (Plus @@ #) / Length[#] &@Partition[x, Length[x] - 2 * m, 1];
```

```
movingAverage[Range[10 000], 10]; // Timing
{0.02, Null}
```

Normally, you don' t get a big penalty in efficiency if you use just one or two global variables for side effects inside the mapped function. But manipulations with entire lists, like those done above, are costly and should be avoided.

▪ 5.2.2.10 More general uses of Map - Mapping on a certain level in an expression

As a third optional argument, Map takes a level specification. This is the standard level specification (see section 1.1.7) used in many *Mathematica* constructs - here it indicates the level(s) on which the function should be mapped. Let me remind that a single integer gives the level up to which the function should be mapped, integer in parentheses means that mapping should affect only that level, and a pair of integers in parentheses indicate a range of levels affected by mapping.

▪ *5.2.2.10.1 Initial examples*

```
Clear[f, testexpr];
```

This is our initial list

```
testexpr = Outer[List, Range[2], Range[3]]
{{{1, 1}, {1, 2}, {1, 3}}, {{2, 1}, {2, 2}, {2, 3}}}
```

First, let us use simple Map:

```
Map[f, testexpr]
{f[{{1, 1}, {1, 2}, {1, 3}}], f[{{2, 1}, {2, 2}, {2, 3}}]}
```

The way to think about Map-ping on higher levels is that we effectively "sneak" through some number of curly braces, and then Map.

Now we Map on levels 1 and 2:

```
Map[f, testexpr, 2]


{f[{f[{1, 1}], f[{1, 2}], f[{1, 3}]}],
 f[{f[{2, 1}], f[{2, 2}], f[{2, 3}]}]}
```

The same can be achieved by

```
Map[f, testexpr, {1, 2}]
{f[{f[{1, 1}], f[{1, 2}], f[{1, 3}]}],
 f[{f[{2, 1}], f[{2, 2}], f[{2, 3}]}]}
```

Now we Map on levels 2 and 3:

```
Map[f, testexpr, {2, 3}]
```

```
{{f[{f[1], f[1]}], f[{f[1], f[2]}], f[{f[1], f[3]}]},
 {f[{f[2], f[1]}], f[{f[2], f[2]}], f[{f[2], f[3]}]}}
```

Now only on level 3:

```
Map[f, testexpr, {3}]
```

```
{{{f[1], f[1]}, {f[1], f[2]}, {f[1], f[3]}},
 {{f[2], f[1]}, {f[2], f[2]}, {f[2], f[3]}}}
```

Negative levels can also be used. In this case negative 1 is equivalent to 3:

```
Map[f, testexpr, {-1}]
```

```
{{{f[1], f[1]}, {f[1], f[2]}, {f[1], f[3]}},
 {{f[2], f[1]}, {f[2], f[2]}, {f[2], f[3]}}}
```

Be aware however that for general expressions (trees, for example), there is no simple connection between negative and positive level specifications. Specification {-n} means "all sub-expressions of depth n", while {n} means all elements at the level <n>. Negative level specifications may look exotic, but in practice they are often quite useful. One particular example where Map-ping on negative levels is useful is when one has to grow some tree in a breadth-first manner.

Specification <-n> without parentheses means "all subexpressions of depth at least n". For example,

```
Map[f, testexpr, -1]
```

```
{f[{f[{f[1], f[1]}], f[{f[1], f[2]}], f[{f[1], f[3]}]}],
 f[{f[{f[2], f[1]}], f[{f[2], f[2]}], f[{f[2], f[3]}]}]}
```

```
Map[f, testexpr, -2]
```

```
{f[{f[{1, 1}], f[{1, 2}], f[{1, 3}]}],
 f[{f[{2, 1}], f[{2, 2}], f[{2, 3}]}]}
```

```
Map[f, testexpr, -3]
```

```
{f[{{1, 1}, {1, 2}, {1, 3}}], f[{{2, 1}, {2, 2}, {2, 3}}]}
```

Once again, for nested lists with the same dimensions of the sublists, we see the connection to positive level specification, because on each level all elements have the same depth. In general, there is no such connection.

- *5.2.2.10.2 Less trivial example - using Map to sort sublists in a nested list*

Create a list (table) of depth 4, of random numbers

```
Clear[testexpr];
testexpr = Table[Random[Integer, {1, 10}], {3}, {2}, {3}]
```

```
{{{1, 7, 9}, {3, 3, 3}}, {{5, 10, 1}, {9, 5, 2}}, {{3, 6, 10}, {8, 8, 3}}}
```

Now say we need to sort the sublists of numbers. The way to do it is to Map the Sort function on the level {2}:

```
Map[Sort, testexpr, {2}]
{{{1, 7, 9}, {3, 3, 3}}, {{1, 5, 10}, {2, 5, 9}}, {{3, 6, 10}, {3, 8, 8}}}
```

If we want to sort in decreasing order, then:

```
Map[Sort[#, #1 ≥  #2 &] &, testexpr, {2}]
{{{9, 7, 1}, {3, 3, 3}}, {{10, 5, 1}, {9, 5, 2}}, {{10, 6, 3}, {8, 8, 3}}}
```

This example is a little tricky and requires some explanation: the point is that Sort itself is a higher order function, which takes as a second optional argument the sorting criteria function. The criteria function depends on 2 variables, and in this case is < #1≥ #2&>. There is no confusion between the first argument of Sort # and the argument #1 from the criteria function, because the latter is completely contained in the former. To clarify this, let us do the same thing without the use of pure functions:

```
Clear[sortF, critF];
critF[x_, y_] := x ≥ y;
sortF[x_List] := Sort[x, critF];
```

Now we can Map sortF instead of Sort

```
Map[sortF, testexpr, {2}]
{{{9, 7, 1}, {3, 3, 3}}, {{10, 5, 1}, {9, 5, 2}}, {{10, 6, 3}, {8, 8, 3}}}
```

The price to pay here is that two auxiliary functions have to be introduced and given names. As it happens, we can use a built-in function GreaterEqual, instead of critF:

```
result = Map[Sort[#, GreaterEqual] &, testexpr, {2}]
{{{9, 7, 1}, {3, 3, 3}}, {{10, 5, 1}, {9, 5, 2}}, {{10, 6, 3}, {8, 8, 3}}}
```

The previous two solutions were presented to show how to do things in the general case, when no built-in criteria function is available. However, whenever the built-in function is available, it is always better to use a built-in.

As a next step in this example, imagine that the sublists also have to be sorted, according to which sublist has a smaller first element. For example, sorting the list {{3,1},{5,6,7},{2,8},{4,6}} should give {{2,8},{3,1},{4,6},{5,6,7}}. The proper (pure) criteria function will be in this case #1[[1]]≤ #2[[1]]&, and we have to Map Sort on the first level now

```
result1 = Map[Sort[#, #1[[1]] ≤ #2[[1]] &] &, result]
{{{3, 3, 3}, {9, 7, 1}}, {{9, 5, 2}, {10, 5, 1}}, {{8, 8, 3}, {10, 6, 3}}}
```

Finally, we may want to reorder the largest sublists, according for instance to a total sum of the numbers contained.

A sub-problem: sum all the numbers contained in a nested list

For example, in the first sublist:

```
sublist = result1[[1]]
{{3, 3, 3}, {9, 7, 1}}
```

The best way to sum all the numbers here is to first use Flatten, to remove internal curly braces and make a list flat. Then use Total on the resulting list:

```
numsum = Total[Flatten[sublist]]
26
```

■

Now that we know how to sum all the numbers, we can convert this knowledge into a pure function (sorting criteria): Total[Flatten[#1-#2]]≥0& (here we use that the sublists have the same structure and can thus be subtracted)

```
result2 = Sort[result1, Total[Flatten[#1 - #2]] ≥ 0 &]
{{{8, 8, 3}, {10, 6, 3}}, {{9, 5, 2}, {10, 5, 1}}, {{3, 3, 3}, {9, 7, 1}}}
```

Notice that in this case there is no Map-ping, since we already operate on the level where we should just simply use Sort.

Let me summarize the goal and the code once again: we are given a list of depth 4 of numbers (think of it as a 3-dimensional grid), which we want to sort according to the following rules: the numbers inside the smallest sublists are sorted in decreasing order. The smallest sublists inside next smallest are sorted such that the sublist with the smallest first element comes first. Finally, the largest sublists are sorted by decreasing total of all the elements in the sublist. The sorting has to be performed starting with the smallest sublists (although in this particular example, the order is irrelevant). Here is the code packaged into a function (variables made local):

```
Clear[sortNested];
sortNested[x_List] := Module[{result, result1, result2},
  result = Map[Sort[#, GreaterEqual] &, x, {2}];
  result1 = Map[Sort[#, #1[[1]] ≤ #2[[1]] &] &, result];
  result2 = Sort[result1, Total[Flatten[#1 - #2]] ≥ 0 &]]
```

Check :

```
sortNested[testexpr]
{{{8, 8, 3}, {10, 6, 3}}, {{9, 5, 2}, {10, 5, 1}}, {{3, 3, 3}, {9, 7, 1}}}
```

Notice that we could have avoided the introduction of auxiliary variables by nesting the code, but then it would become much less readable:

```
Clear[sortNestedAlt];
sortNestedAlt[x_List] := Sort[ Map[Sort[#, #1[[1]] ≤ #2[[1]] &] &,
    Map[Sort[#, GreaterEqual] &, x, {2}]],
   Total[Flatten[#1 - #2]] ≥ 0 &];
```

```
sortNestedAlt[testexpr]
{{{8, 8, 3}, {10, 6, 3}}, {{9, 5, 2}, {10, 5, 1}}, {{3, 3, 3}, {9, 7, 1}}}
```

The lesson here is that often the advantage of readability overweights extra code needed. Try to resist a temptation of writing deeply nested functions like <sortNestedAlt>.

A general feature illustrated by this example is that pure functions, Map and a possibility to use higher-order functions greatly reduce the size of the code, and increase its flexibility. Also, in *Mathematica* this way of doing it will be among the most efficient ones.

Many more examples of Map in action will follow - Map is truly ubiquitous in *Mathematica* programming.

5.2.3 MapAt

If Map can be thought of as a machine gun, then MapAt is a precision rifle. It Maps the function to specific position(s) rather than on an entire list.

In the simplest form, MapAt takes 3 arguments: the function to be mapped, the expression, and the position of the element of this expression on which to map. For mapping on the first-level elements of the list, the position can be just the index of the element - a number. In general, the position has to be a list of indices. MapAt uses the same position specifications as Position and Extract. One can also use MapAt to map a function on several elements at once - in this case, a list of positions of these elements, rather than a single position, has to be supplied. The elements corresponding to these positions, may be at different levels in the expression. Thus, sublists representing elements' positions can have different lengths.

5.2.3.1 Examples

```
Clear[ourlist];
ourlist = {a, b, c, d}
```

```
{a, b, c, d}
```

Say, we want to Map Sine function on the second element:

```
MapAt[Sin, ourlist, 2]
```

```
{a, Sin[b], c, d}
```

If we want to Map on second and third elements:

```
MapAt[Sin, ourlist, {{2}, {3}}]
```

```
{a, Sin[b], Sin[c], d}
```

Notice the internal curly braces. If we omit them, *Mathematica* will decide that we want to Map Sine on a single element with the position {2,3}. Since there is no such element ({2,3} means third element of the second element, but the second element is an atom), this will result in an error.

```
MapAt[Sin, ourlist, {2, 3}]
```

```
MapAt::partw : Part {2, 3} of {a, b, c, d} does not exist. More…
MapAt[Sin, {a, b, c, d}, {2, 3}]
```

Let us now create a nested list:

```
Clear[testexpr, f];
testexpr = Table[Random[Integer, {1, 10}], {3}, {2}, {3}]
```

```
{{{3, 1, 7}, {3, 4, 10}}, {{2, 7, 3}, {1, 4, 5}}, {{3, 8, 2}, {1, 2, 3}}}
```

It is useful to consider a few examples and try to understand why each output is as it is:

```
MapAt[f, testexpr, 2]
```

```
{{{3, 1, 7}, {3, 4, 10}},
 f[{{2, 7, 3}, {1, 4, 5}}], {{3, 8, 2}, {1, 2, 3}}}
```

```
MapAt[f, testexpr, {{2}, {3}}]
```

```
{{{3, 1, 7}, {3, 4, 10}},
 f[{{2, 7, 3}, {1, 4, 5}}], f[{{3, 8, 2}, {1, 2, 3}}]}
```

```
MapAt[f, testexpr, {3, 2}]
```

```
{{{3, 1, 7}, {3, 4, 10}},
 {{2, 7, 3}, {1, 4, 5}}, {{3, 8, 2}, f[{1, 2, 3}]}}
```

```
MapAt[f, testexpr, {{2, 1}, {3, 2}}]
```

```
{{{3, 1, 7}, {3, 4, 10}},
 {f[{2, 7, 3}], {1, 4, 5}}, {{3, 8, 2}, f[{1, 2, 3}]}}
```

```
MapAt[f, testexpr, {{2, 1, 1}, {3, 2, 3}}]
```

```
{{{3, 1, 7}, {3, 4, 10}},
 {{f[2], 7, 3}, {1, 4, 5}}, {{3, 8, 2}, {1, 2, f[3]}}}
```

Basically, all you have to do to unravel the above examples is to carefully trace the position: for example, {2,1,1} means: first element of the first element of the second element.

- 5.2.3.2 Use in conjunction with Map

MapAt is often used in conjunction with Map. For example, we want to Map <f> on the second element (number) in each small sublist. The code is

```
Map[MapAt[f, #, 2] &, testexpr, {2}]
```

```
{{{3, f[1], 7}, {3, f[4], 10}},
 {{2, f[7], 3}, {1, f[4], 5}}, {{3, f[8], 2}, {1, f[2], 3}}}
```

To convince yourself that we needed to Map on the level {2}, think this way: we had to bypass 2 external curly braces to use MapAt on the first sublist. Should we Map on the first level instead, we' d get the following:

```
Map[MapAt[f, #, 2] &, testexpr]
```

```
{{{3, 1, 7}, f[{3, 4, 10}]},
 {{2, 7, 3}, f[{1, 4, 5}]}, {{3, 8, 2}, f[{1, 2, 3}]}}
```

which is not what we want (do you see what happened and why?).Notice that we created the pure function out of MapAt along the lines of section 4.11.1.12.

5.2.3.3 Use in conjunction with Position

Also, MapAt is often used in conjunction with Position. The logic is this: say, we want to Map certain function f on all elements of a given expression(possibly restricted to some levels), satisfying certain criteria. This can be easily done in a rule-based approach. In FP approach, we use Position to find positions of all such elements, and then plug the result into MapAt.

As an example, say we want to Map f on all even numbers in our list. The code is

```
MapAt[f, testexpr, Position[testexpr, _?EvenQ]]
{{{3, 1, 7}, {3, f[4], f[10]}},
 {{f[2], 7, 3}, {1, f[4], 5}}, {{3, f[8], f[2]}, {1, f[2], 3}}}
```

Or, on all multiples of 3

```
MapAt[f, testexpr, Position[testexpr, x_ /; Mod[x, 3] == 0]]
{{{f[3], 1, 7}, {f[3], 4, 10}},
 {{2, 7, f[3]}, {1, 4, 5}}, {{f[3], 8, 2}, {1, 2, f[3]}}}
```

Warning : performance pitfall

However, be aware of the performance pitfall associated with this technique: when you try to use MapAt to map a function on a large number of elements at the same time (especially if all of them belong to a single subexpression), MapAt can be very slow. Please see Appendix C for a detailed discussion. In some cases, the performance of MapAt may be improved (or, rather, a more efficient implementation with a basic MapAt functionality can be found) - see chapter VI for such an example.

5.2.3.4 Application : multiple random walks

We can use MapAt to implement co - evolving random walkers, for instance. Let us say that each walker is updated with the same function, say

```
Clear[randomWalk];
randomWalk[x_] := x + Random[Integer, {-1, 1}];
```

Let us say we have n = 5 random walkers, all starting at zero :

```
nwalkers = 5;
startpositions = Table[0, {nwalkers}]
{0, 0, 0, 0, 0}
```

We may either update them one by one, or in principle, update them at random. In the latter case, we will generate an "update" list of numbers from 1 to n. Each number < k > means "update walker k".

```
totalupdates = 50;

updatelist = Table[Random[Integer, {1, nwalkers}], {totalupdates}]
{1, 4, 5, 2, 3, 3, 1, 4, 4, 5, 2, 2, 2, 2, 2, 2, 1, 2, 4, 2, 3, 4, 3, 3, 2,
 1, 4, 3, 3, 2, 1, 1, 1, 2, 2, 2, 1, 5, 5, 2, 4, 4, 1, 5, 3, 2, 3, 2, 1, 2}
```

This gives the total evolution of all the walkers (FoldList will be covered shortly) :

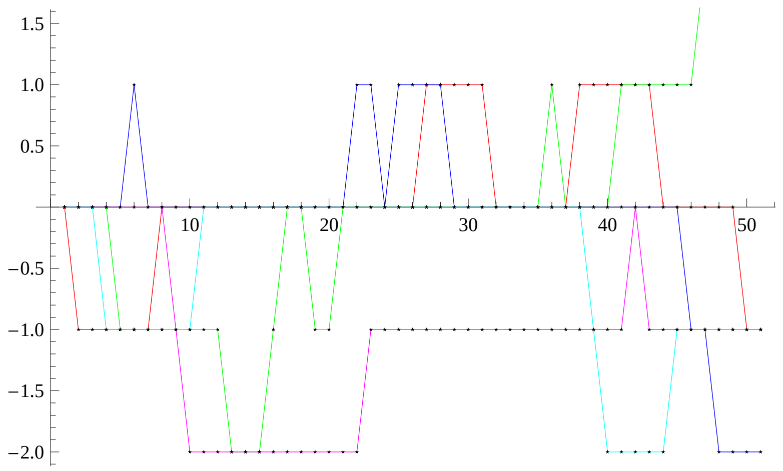
```
Short[result = FoldList[
   MapAt[randomWalk, #1, #2] &, startpositions, updatelist], 10]
```

```
{{0, 0, 0, 0, 0}, {-1, 0, 0, 0, 0}, {-1, 0, 0, 0, 0},
 {-1, 0, 0, 0, -1}, {-1, -1, 0, 0, -1}, {-1, -1, 1, 0, -1},
 {-1, -1, 0, 0, -1}, ≪37≫, {0, 1, 0, -1, -1},
 {0, 1, -1, -1, -1}, {0, 2, -1, -1, -1}, {0, 2, -2, -1, -1},
 {0, 3, -2, -1, -1}, {-1, 3, -2, -1, -1}, {-1, 4, -2, -1, -1}}
```

To get individual walker' s trajectories, we may Transpose the result. We will us the < MultipleListPlot > package to visualize the walks

```
Needs["Graphics‘MultipleListPlot‘"];
```

```
MultipleListPlot[Transpose[result], PlotJoined → True,
 PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 1, 0],
   RGBColor[0, 0, 1], RGBColor[1, 0, 1], RGBColor[0, 1, 1]},
 SymbolShape → PlotSymbol[Star, 1]]
```



- 5.2.3.5 Identical positions in the position list

When some of the positions in the position list in MapAt coincide, the function is mapped on that position several times (nested). For instance :

```
Clear[g];
MapAt[g, Range[5], {{1}, {2}, {4}, {2}, {4}, {4}}]
```

```
{g[1], g[g[2]], 3, g[g[g[4]]], 5}
```

However, the order in which the function is mapped in this case does not correspond to the order of positions in the list of positions. The nested function calls which correspond to the duplicate positions in the position list, are grouped together before the function is evaluated. We can see that by introducing a side - effect - the counter n, and trace whether the counter values correspond to the order in which mapping positions follow - they don' t.

```
MapAt[g[#, n ++] &, n = 0; Range[5], {{1}, {2}, {4}, {2}, {4}, {4}}]
```

```
{g[1, 0], g[g[2, 1], 2], 3, g[g[g[4, 3], 4], 5], 5}
```

This means, in particular, that in the previous example with random walkers we won' t be able to obtain the final state of the walkers by just feeding the list of update positions to MapAt, because it would apply the random < randomWalk > function to the walkers in different order.

5.2.3.6 The order of mapping

Whenever MapAt is used to map a function to a number of elements, we already saw that MapAt does not map in the same order in which we supply the positions of the element. But what is the order in which mapping happens? The answer is that the list of positions is first reordered such that they follow as they would in a depth - first expression traversal. The reason is probably that this is the only order which is unambiguous, given that the function being mapped may in principle do any transformation with the subexpression (sub - branch) on which it is mapped, even destroy it.

5.2.3.7 Example: imitating DeleteCases

One important specific case is when we want to actually delete elements subject to some criteria. Usually, we use DeleteCases for that. For example, let us delete from all sublists of a nested list all elements that are multiples of 3. This will be our test list:

```
Clear[testexpr];
testexpr = Table[Random[Integer, {1, 10}], {3}, {2}, {3}]
```

```
{{{8, 6, 6}, {3, 7, 1}}, {{3, 2, 5}, {7, 9, 8}}, {{8, 10, 6}, {6, 5, 3}}}
```

We can imitate DeleteCases by Mapping a function #/.#:→Sequence[]&, which effectively deletes an element from the list by replacing it by an "emptiness" (if you are feeling ambitious, try to see why we did not use a simpler form for the function being mapped - just Sequence[]&).

```
MapAt[# /. # :→ Sequence[] &, testexpr,
 Position[testexpr, x_ /; Mod[x, 3] == 0]]
```

```
{{{8}, {7, 1}}, {{2, 5}, {7, 8}}, {{8, 10}, {5}}}
```

We can package this into a function :

```
Clear[myDeleteCases];
myDeleteCases[expr_, patt_] :=
  MapAt[# /. # :→ Sequence[] &, expr, Position[expr, patt]];
```

Check again :

```
myDeleteCases[testexpr, x_ /; Mod[x, 3] == 0]
```

```
{{{8}, {7, 1}}, {{2, 5}, {7, 8}}, {{8, 10}, {5}}}
```

With DeleteCases, it will look like (Infinity means look at all levels)

```
DeleteCases[testexpr, x_ /; Mod[x, 3] == 0, Infinity]
```

```
{{{8}, {7, 1}}, {{2, 5}, {7, 8}}, {{8, 10}, {5}}}
```

5.2.4 MapAll

This function is equivalent to **Map[function,expression,{0,Infinity}]**. The zero here is necessary to apply the function also to entire expression, since this is what MapAll also does. We give just a single example

```
Clear[testexpr, f];
testexpr = Table[Random[Integer, {1, 10}], {3}, {2}, {3}]
{{{7, 4, 7}, {10, 7, 6}}, {{9, 7, 6}, {1, 7, 4}}, {{1, 7, 1}, {7, 7, 6}}}
MapAll[f, testexpr]
f[{f[{f[{f[7], f[4], f[7]}], f[{f[10], f[7], f[6]}]}],
  f[{f[{f[9], f[7], f[6]}], f[{f[1], f[7], f[4]}]}],
  f[{f[{f[1], f[7], f[1]}], f[{f[7], f[7], f[6]}]}]}]
```

5.2.4.1 MapAll works in a depth-first manner

As is easy to demonstrate, MapAll performs the mapping on a general *Mathematica* expression (which, as we recall, can always be represented as a tree), in a depth - first manner. To do this, consider the following nested list (Fold operation will be covered shortly):

```
tst = Fold[List, {}, {a, b, c, d}]
{{{{{}, a}, b}, c}, d}
```

This function prints the value of its argument and then returns the argument.

```
ShowIt[x_] := (Print[x]; x)
```

Let us now map it to all levels of our expression :

```
MapAll[ShowIt, tst]
{}
a
{{}, a}
b
{{{}, a}, b}
c
{{{{}, a}, b}, c}
d
{{{{{}, a}, b}, c}, d}
{{{{{}, a}, b}, c}, d}
```

This clearly demonstrates that Mapping is performed in a depth - first manner. In general, MapAll should be used anytime when some function has to be applied to all levels of expression and in the depth-first manner (or when the order in which the function is applied to different pieces of expression does not matter). One example of such use follows.

5.2.4.2 Use in conjunction with ReplaceAll

MapAll is not very often used, but there is however at least one instance when it is very useful - in conjunction with rule application when rules have to be applied to the innermost parts of the expression first. This is not what happens by default. Consider the following rule :

```
ourrule = {x_, y_} :> {x, y^2};
```

Now, let us try :

```
tst /. ourrule
```

$\{\{\{\{\{\}, a\}, b\}, c\}, d^2\}$

```
tst /. ourrule /. ourrule
```

$\{\{\{\{\{\}, a\}, b\}, c\}, d^4\}$

We see that the rule is always being applied to the outermost expression, since it matches. One way to proceed is of course to restrict the rule, so that, once applied, it won't apply to the transformed expression any more :

```
ournewrule = {x_, y_ /; Not[MatchQ[y, Power[_, 2]]]} :> {x, y^2};
tst /. ournewrule
```

$\{\{\{\{\{\}, a\}, b\}, c\}, d^2\}$

```
tst /. ournewrule /. ournewrule
```

$\{\{\{\{\{\}, a\}, b\}, c^2\}, d^2\}$

```
tst //. ournewrule
```

$\{\{\{\{\{\}, a^2\}, b^2\}, c^2\}, d^2\}$

In the above case, we had our rule applied to expression starting from the outermost level to the innermost. But this behavior is not always the desired one. We can use MapAll to change this : now we will do the same by starting from the innermost piece, since MapAll maps the function in a depth - first manner. We can monitor this by tracing the execution (we restrict the Trace command to show only the rule-substitution pieces of the evaluation tree)

```
Trace[MapAll[# /. ourrule &, tst], ReplaceAll]
{{{{{{{{{{} /. ourrule, {} /. {x_, y_} :→ {x, y^2}, {}},
        {a /. ourrule, a /. {x_, y_} :→ {x, y^2}, a}},
       {{}, a} /. ourrule, {{}, a} /. {x_, y_} :→ {x, y^2}, {{}, a^2}},
      {b /. ourrule, b /. {x_, y_} :→ {x, y^2}, b}}, {{{}, a^2}, b} /. ourrule,
     {{{}, a^2}, b} /. {x_, y_} :→ {x, y^2}, {{{}, a^2}, b^2}},
    {c /. ourrule, c /. {x_, y_} :→ {x, y^2}, c}},
   {{{{}, a^2}, b^2}, c} /. ourrule,
   {{{{}, a^2}, b^2}, c} /. {x_, y_} :→ {x, y^2},
   {{{{}, a^2}, b^2}, c^2}},
  {d /. ourrule, d /. {x_, y_} :→ {x, y^2}, d}},
 {{{{{}, a^2}, b^2}, c^2}, d} /.
  ourrule,
 {{{{{}, a^2}, b^2}, c^2}, d} /.
  {x_, y_} :→ {x, y^2},
 {{{{{}, a^2}, b^2}, c^2}, d^2}}
```

■ A digression : performance improvements and the use of With

Trace also reveals that our computation is not entirely efficient since the rule definition is evaluated every time afresh. To avoid this we will use < With > to embed the rule definition textually :

```
Trace[With[{rule = ourrule}, MapAll[# /. rule &, tst]], ReplaceAll]
{{{{{{{{{{} /. {x_, y_} :→ {x, y^2}, {}}, {a /. {x_, y_} :→ {x, y^2}, a}},
       {{}, a} /. {x_, y_} :→ {x, y^2}, {{}, a^2}},
      {b /. {x_, y_} :→ {x, y^2}, b}}, {{{}, a^2}, b} /. {x_, y_} :→ {x, y^2},
     {{{}, a^2}, b^2}}, {c /. {x_, y_} :→ {x, y^2}, c}},
   {{{{}, a^2}, b^2}, c} /. {x_, y_} :→ {x, y^2}, {{{{}, a^2}, b^2}, c^2}},
  {d /. {x_, y_} :→ {x, y^2}, d}},
 {{{{{}, a^2}, b^2}, c^2}, d} /.
  {x_, y_} :→ {x, y^2},
 {{{{{}, a^2}, b^2}, c^2}, d^2}}
```

Note that the use of Module or Block won’t have the same effect, and won’t help us here.

■

The same functionality can be achieved by using Replace rather than ReplaceAll, with the level specification {0, Infinity} :

```
Trace[Replace[tst, ourrule, {0, Infinity}]]
{{tst, {{{{{}, a}, b}, c}, d}},
 {ourrule, {x_, y_} :→ {x, y^2}}, {{∞, ∞}, {0, ∞}},
 Replace[{{{{{}, a}, b}, c}, d}, {x_, y_} :→ {x, y^2}, {0, ∞}],
 {{{{{}, a^2}, b^2}, c^2}, d^2}}
```

This is in fact a more efficient solution since more operations are done inside the kernel, but without the previous discussion the apparent differences in the functionality of ReplaceAll[expr, rules] and Replace[expr, rules, {0, Infinity}] may look like a mystery. In many cases the desired behavior is in fact the one of the latter operation - that is, apply rules to the innermost expressions first.

Also, even in the case above, the use of the construct MapAll[#/.rules&,expr] may be advantageous since, being essentially equivalent to Replace[expr, rules, {0, Infinity}], it allows for a more detailed tracing and debugging. Once the code is tested (and possibly debugged), one can change it back to Replace[expr, rules, {0, Infinity}].

Finally, it could be necessary that the rules at each level be repeatedly applied until the expression stops changing. This is achieved by MapAll[ReplaceRepeated[#,rules]&,expr], but to my knowledge there is no simple equivalent with Replace in this case.

5.2.5 Scan

Scan is a function similar to Map, but it does not return a list of values at the end. This means that using Scan only makes sense if the function being scanned contains side effects such as assignments. The format is :

Scan[f, expr, levspec],

where <f> is the function being scanned, <expr> is an expression on which we scan the function <f>, and <levspec> is an optional level specification - the syntax is very similar to that for Map.

5.2.5.1 Simple examples

Let us, for example, scan a squaring function on a list :

```
Scan[#^2 &, Range[10]]
```

We can see that no output has been produced (or, more precisely, Null has been produced). To get anything useful, we need some side effects. Here, for instance, we will use Scan to compute the sum of squares of the list elements:

```
Module[{sumsq = 0}, Scan[sumsq += #^2 &, Range[10]];
 sumsq]
```

```
385
```

Since Scan does not produce a list of results, it is somewhat faster than Map. However, there is another and perhaps more important difference between the two : Scan can be "stopped" at any moment by using a

Return statement inside a function being scanned. This is not true for Map - it can be stopped only by throwing an exception. For example, we want to compute the sum of the squares but stop as the element exceeds 6 :

```
Module[{sumsq = 0},
 Scan[If[# ≤ 6, sumsq += #^2, Return[sumsq]] &, Range[10]]]
91
```

The Return statement will break only from Scan, but not from any scoping construct possibly enclosing Scan, such as Module above.

5.2.5.2 Example: conditional list splitting

Scan can also be thought of as a replacement for loops. Here, for example, we will use it to determine the position where to split a given list in 2 parts (this will happen as soon as the condition < cond > will first be satisfied).

```
Clear[splitWhen];
splitWhen[x_List, cond_] := Module[{n = 0},
  Scan[If[cond[#], Return[], n ++] &, x]; {Take[x, n], Drop[x, n]}]
```

Scan is a better device than just a single loop (or nested loops in most cases, for that matter), both because it is optimized and because it is more general: it receives the standard level specification and works on general symbolic trees (*Mathematica* expressions), not just simple lists . In particular, if asked, it traverses an expression depth - first, computing whatever side effects we instruct it to.

5.2.6 MapIndexed

This is a truly useful function, which extends in some sense the capabilities of Map. There are situations in which on one hand, a function like Map is needed, but on the other hand, which Map can not handle. These are cases when the function being mapped has to "know" where in the list it is "currently". Let us consider an example.

5.2.6.1 Starting example and syntax

Say, we have a simple list of numbers:

```
testlist = Table[Random[Integer, {0, 10}], {15}]
{7, 1, 7, 2, 6, 2, 2, 0, 2, 5, 8, 7, 8, 7, 7}
```

Now, say, we would like to Map on it a function (-1)^n*Sin[x], where <n> is a position of the number, and <x> is a number. In this simple case we could use Map like this:

```
Module[{n = 0},
 Map[(-1)^(n ++) Sin[#] &, testlist]]

{Sin[7], -Sin[1], Sin[7], -Sin[2], Sin[6], -Sin[2], Sin[2], 0,
 Sin[2], -Sin[5], Sin[8], -Sin[7], Sin[8], -Sin[7], Sin[7]}
```

However, this is not an aesthetic solution. Also, it will not work for more complicated lists. What MapIndexed does is to provide to a function being mapped the position of the current element as a second argument. So, now the function being mapped is a function of two arguments. The rest of the syntax is the same:

MapIndexed[function,expression,level]

As before, the level specification is an optional parameter - it is 1 by default. In this particular example, we write

```
MapIndexed[(-1)^#2[[1]]*Sin[#1] &, testlist]
```

```
{-Sin[7], Sin[1], -Sin[7], Sin[2], -Sin[6], Sin[2], -Sin[2], 0,
 -Sin[2], Sin[5], -Sin[8], Sin[7], -Sin[8], Sin[7], -Sin[7]}
```

Notice that we again use here a pure function, but this time a pure function of the two arguments. Notice also that we take a first part of the second argument #2[[1]]: this is because the position is always given as a list of indexes, even for a simple list.

In principle, once again, we can use a pattern-defined function of two arguments here:

```
Clear[g];
g[x_, {n_Integer}] := (-1)^n*Sin[x];
MapIndexed[g, testlist]
```

```
{-Sin[7], Sin[1], -Sin[7], Sin[2], -Sin[6], Sin[2], -Sin[2], 0,
 -Sin[2], Sin[5], -Sin[8], Sin[7], -Sin[8], Sin[7], -Sin[7]}
```

Another simple example: let us supply the numbers in the list with their positions

```
MapIndexed[List, testlist]
```

```
{{4, {1}}, {0, {2}}, {10, {3}}, {2, {4}}, {7, {5}},
 {7, {6}}, {1, {7}}, {10, {8}}, {8, {9}}, {9, {10}},
 {6, {11}}, {6, {12}}, {5, {13}}, {8, {14}}, {3, {15}}}
```

■ 5.2.6.2 More examples

■ 5.2.6.2.1 Example: creation of specific matrices

MapIndexed gets more non-trivial in what can be accomplished with it, as we go to nested lists and trees. For example, let us build a matrix (list of lists), with elements Sin[i-j], where i and j are column and row numbers (say,4x4)

```
(result = MapIndexed[Sin[#2[[1]] -#2[[2]]] &,
    IdentityMatrix[4], {2}]) // MatrixForm
```

```
⎛   0     -Sin[1]  -Sin[2]  -Sin[3] ⎞
⎜ Sin[1]     0     -Sin[1]  -Sin[2] ⎟
⎜ Sin[2]   Sin[1]     0     -Sin[1] ⎟
⎝ Sin[3]   Sin[2]   Sin[1]     0    ⎠
```

Notice that here we Map on the level {2}, which corresponds to numbers. Also, then, the argument #2 is

Notice that here we Map on the level {2}, which corresponds to numbers. Also, then, the argument #2 is now a position of the matrix element and has 2 indices (i and j). It is really easy to verify that - create a matrix where the elements will be just positions:

```
MapIndexed[#2 &, IdentityMatrix[4], {2}]
{{{1, 1}, {1, 2}, {1, 3}, {1, 4}}, {{2, 1}, {2, 2}, {2, 3}, {2, 4}},
 {{3, 1}, {3, 2}, {3, 3}, {3, 4}}, {{4, 1}, {4, 2}, {4, 3}, {4, 4}}}
```

Now let us say we want to Map a value <a> on the two diagonals above and below the main diagonal. Here is the code:

```
(result1 = MapIndexed[
    If[Abs[#2[[1]] -#2[[2]]] == 1, a, #1] &, result, {2}]) // MatrixForm
```

$$\begin{pmatrix} 0 & a & -\mathrm{Sin}[2] & -\mathrm{Sin}[3] \\ a & 0 & a & -\mathrm{Sin}[2] \\ \mathrm{Sin}[2] & a & 0 & a \\ \mathrm{Sin}[3] & \mathrm{Sin}[2] & a & 0 \end{pmatrix}$$

Not that in this particular case, MapIndexed - based solution is not very efficient, since since the majority of matrix element tested by MapIndexed were a priori known not to be on the diagonals of interest. Later in this chapter we will give a more efficient solution based on MapThread function.

Note also that in principle this example can be generalized to create any matrix we want. All we need is to define a function which will compute a matrix element given its position in the matrix. This function will then be used by MapIndexed. Also, note that we had to start with some matrix (here we used IdentityMatrix).

- 5.2.6.2.2 Example: creation and manipulation of matrices of functions

We can do more interesting things. In particular, we can construct a matrix where all elements will be functions:

```
(result3 = MapIndexed[x^Total[#2] &, IdentityMatrix[3], {2}]) //
 MatrixForm
```

$$\begin{pmatrix} x^2 & x^3 & x^4 \\ x^3 & x^4 & x^5 \\ x^4 & x^5 & x^6 \end{pmatrix}$$

We may compute the determinant symbolically:

```
Det[result3]
0
```

We can , say, differentiate the off-diagonal elements and integrate the diagonal elements:

```
(result4 = MapIndexed[
    Simplify[If[#2[[1]] == #2[[2]], Integrate[#1, x], D[#1, x]]] &,
    result3, {2}]) // MatrixForm
```

$$\begin{pmatrix} \frac{x^3}{3} & 3\,x^2 & 4\,x^3 \\ 3\,x^2 & \frac{x^5}{5} & 5\,x^4 \\ 4\,x^3 & 5\,x^4 & \frac{x^7}{7} \end{pmatrix}$$

- 5.2.6.2.3 Example: imitating the Position command

One can imitate the Position operation with MapIndexed:

```
Clear[testexpr];
testexpr = Table[Random[Integer, {1, 10}], {3}, {2}, {3}]


{{{5, 1, 3}, {5, 5, 1}}, {{5, 3, 2}, {4, 8, 7}}, {{3, 2, 4}, {8, 8, 2}}}
```

Let us find positions of all elements divisible by 3:

```
Flatten[MapIndexed[
  If[Mod[#1, 3] == 0, #2, #1 /. #1 :> Sequence[]] &, testexpr, {3}], 2]
{{1, 1, 3}, {2, 1, 2}, {3, 1, 1}}
Clear[g, result, result1, result2, result3, result4, testexpr];
```

- 5.2.6.2.4 Example: imitating a Partition command

Say we have a list of numbers, and would like to partition it into some sublists (without overlaps for simplicity). This is normally done by a Partition command, but we may try to imitate it by MapIndexed.

```
Clear[testlist];
testlist = Table[ Random[Integer, {1, 10}], {15}]
{1, 1, 4, 8, 9, 9, 10, 3, 8, 7, 1, 8, 10, 8, 9}
```

Say we want to partition this into a sublist of 4 elements each (the last 3 elements will be lost then). First, create the proper list structure:

```
struct = Table[0, {IntegerPart[15 / 4]}, {4}]
{{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}
```

Now use MapIndexed

```
MapIndexed[testlist[[#2[[2]] + (#2[[1]] -1) * 4]] &, struct, {2}]
{{1, 1, 4, 8}, {9, 9, 10, 3}, {8, 7, 1, 8}}
```

We can now package this into a function:

```
Clear[myPartition];
myPartition[lst_List, size_Integer] :=
 Module[{len = Length[lst], struct},
  struct  = Table[0, {IntegerPart[len / size]}, {size}];
  MapIndexed[lst[[#2[[2]] + (#2[[1]] - 1) * size]] &, struct, {2}]]
```

Test:

```
myPartition[testlist, 4]
{{1, 1, 4, 8}, {9, 9, 10, 3}, {8, 7, 1, 8}}
myPartition[testlist, 3]
{{1, 1, 4}, {8, 9, 9}, {10, 3, 8}, {7, 1, 8}, {10, 8, 9}}
myPartition[Range[10], 2]
{{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}}
```

It is interesting to compare the performance of our version vs. the built-in:

```
myPartition[Range[400 000], 3]; // Timing
{5.398 Second, Null}
Partition[Range[400 000], 3]; // Timing
{0.04 Second, Null}
```

As expected, we are not even close (difference more than a hundred times on my machine), although there are definitely ways to do it even much worse than we did. On the practical side, this once again confirms the rule: use built-in functions whenever possible, and design the programs so.

■ 5.2.6.2.5 Example: computing an unsorted union of a list

Here we will use MapIndexed to create a set of rules. The problem will be to compute an unsorted Union of a list of objects. To remind, Union operation returns a sorted list of all distinct elements of an input list (removes duplicates plus sorts, see section 3.10.2). For example:

```
Union[{b, c, a, d, c, d, a, c, b}]
{a, b, c, d}
```

We now want our <unsortedUnion> function to also remove the duplicates but not to sort the resulting list. For instance, for the previous input, the answer should be

```
{b, c, a, d}
```

Our present implementation will be based on application of rules. There is an elegant alternative implementation with the Reap-Sow technique, but this we will discuss later.

The idea here will be the following: we can first create a list of rules in the form {element1→position1,...}. Then we will use the standard Union to get the sorted union of the input list. We will then apply the rules to it, to get a list of positions where these elements are first present in the list (since in the case of a list of rules, only the first rule that matches an element is applied, and then rules are applied to the next element)

. We will get then a list of positions. What remains is to Sort this list and then extract the corresponding elements. So, let us now do this step by step:

This will be our test list:

```
testlist = Table[Random[Integer, {1, 10}], {20}]
{1, 4, 5, 8, 2, 5, 1, 3, 6, 3, 6, 3, 6, 2, 7, 2, 1, 7, 10, 2}
```

Now, we will use MapIndexed to create a set of rules:

```
rules = MapIndexed[Rule, testlist]
{1 → {1}, 4 → {2}, 5 → {3}, 8 → {4}, 2 → {5}, 5 → {6}, 1 → {7},
 3 → {8}, 6 → {9}, 3 → {10}, 6 → {11}, 3 → {12}, 6 → {13}, 2 → {14},
 7 → {15}, 2 → {16}, 1 → {17}, 7 → {18}, 10 → {19}, 2 → {20}}
```

Let us now compute the Union and apply the rules:

```
un = Union[testlist]
{1, 2, 3, 4, 5, 6, 7, 8, 10}
un /. rules
{{1}, {5}, {8}, {2}, {3}, {9}, {15}, {4}, {19}}
```

we now Sort this list:

```
Sort[un /. rules]
{{1}, {2}, {3}, {4}, {5}, {8}, {9}, {15}, {19}}
```

All that remains is to Extract the elements:

```
Extract[testlist, Sort[un /. rules]]
{1, 4, 5, 8, 2, 3, 6, 7, 10}
```

We can now combine everything together:

```
Clear[unsortedUnion];
unsortedUnion[x_List] :=
  Extract[x, Sort[Union[x] /. Dispatch[MapIndexed[Rule, x]]]];
```

The Dispatch command will be covered later. For now, let me just say that this command makes the rule application more efficient, by hashing together the rules which can not apply simultaneously. This is particularly relevant for our present function, since all the rules for duplicate elements will be optimized with Dispatch. Once we cover Dispatch, we will revisit this problem and make a performance test to see how much we gain from using Dispatch. For now, let us just check that the function works correctly:

```
unsortedUnion[testlist]
{1, 4, 5, 8, 2, 3, 6, 7, 10}
Clear[testlist, list, un, unsortedUnion];
```

5.2.6.2.6 Example: computing frequencies of objects in a list

The technique based on the combination of MapIndexed , Dispatch and Union, used in the previous example, can be used also to compute frequencies of the objects in a list (I remark that in versions prior to 6.0 this function can be found in Statistics‘DataManipulation package and is implemented with the use of Split command - we covered this implementation in section 3.10.3.4. In version 6.0, the function Tally takes over this functionality, and then of course should be used since it is faster).

Let us develop the <frequencies> function. Here is our test list :

```
testlist = Table[Random[Integer, {1, 20}], {40}]
```

```
{11, 17, 16, 6, 8, 12, 17, 3, 18, 1, 13, 14, 2, 15, 19, 16, 19, 11, 4, 4,
 8, 2, 19, 2, 8, 15, 6, 14, 12, 5, 18, 12, 14, 1, 16, 7, 12, 15, 17, 18}
```

The first step will be the same as before: create a set of rules <element -> position> for the Union of elements in the list, and then use these rules to replace all the elements in the initial list with these positions:

Here is the (sorted) Union of our list

```
un = Union[testlist]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 16, 17, 18, 20}
```

Here is a set of rules

```
rules = MapIndexed[Rule[#1, First[#2]] &, un]
```

```
{1 → 1, 2 → 2, 3 → 3, 4 → 4, 5 → 5, 6 → 6, 7 → 7, 8 → 8, 9 → 9, 11 → 10,
 12 → 11, 13 → 12, 15 → 13, 16 → 14, 17 → 15, 18 → 16, 20 → 17}
```

Here we have replaced all the elements by their positions, using the Dispatch - ed version of the rules.

```
replaced = ReplaceAll[testlist, Dispatch[rules]]
```

```
{10, 8, 3, 7, 6, 4, 16, 5, 11, 3, 8, 15, 14, 3, 2, 16, 7, 16, 12, 15, 7,
 15, 1, 7, 16, 1, 2, 14, 7, 5, 13, 16, 1, 14, 12, 9, 5, 17, 11, 17}
```

Now, the idea is to create an array of counters

```
counters = Array[0 &, {Length[un]}]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

and Scan the increment of the counter with a given position onto the list we created in the previous step :

```
Scan[counters[[#]] ++ &, replaced]
```

Now we have counted all the distinct objects :

```
counters
```

```
{2, 2, 1, 4, 3, 6, 2, 3, 2, 5, 1, 2, 2, 1, 2, 1, 1}
```

All that is left is to group together the elements of the < un > and respective frequencies. This can be easily accomplished by another MapIndexed, given that the frequency of a given element is contained in the array < counters > at the same position as position of this element in < un >

```
MapIndexed[{#1, counters[[#2[[1]]]]} &, un]
```

```
{{2, 2}, {3, 2}, {4, 1}, {6, 4}, {7, 3}, {8, 6}, {9, 2}, {10, 3}, {11, 2},
 {12, 5}, {13, 1}, {14, 2}, {15, 2}, {16, 1}, {17, 2}, {18, 1}, {19, 1}}
```

We now package everything into a function :

```
Clear[frequencies];
frequencies[x_List] := With[{un = Union[x]},
   Module[{counters = Array[0 &, {Length[un]}]},
    Scan[counters[[#]] ++ &,
      ReplaceAll[x,
       Dispatch[MapIndexed[Rule[#1, First[#2]] &, un]]]];
    MapIndexed[{#1, counters[[#2[[1]]]]} &, un]]];
```

Test :

```
frequencies[testlist]
```

```
{{2, 2}, {3, 2}, {4, 1}, {6, 4}, {7, 3}, {8, 6}, {9, 2}, {10, 3}, {11, 2},
 {12, 5}, {13, 1}, {14, 2}, {15, 2}, {16, 1}, {17, 2}, {18, 1}, {19, 1}}
```

For comparison, here is the implementation of frequencies function from the Statistics‘DataManipulation‘ package.

```
Clear[frequenciesAlt];
frequenciesAlt[x_List] :=
 Map[{First[#], Length[#]} &, Split[Sort[x]]]
```

Let us compare the performance :

```
testlist = Table[Random[Integer, {1, 200}], {1000}];
frequencies[testlist]; // Timing
frequenciesAlt[testlist]; // Timing
```

```
{0.02, Null}

{0.01, Null}
```

```
testlist = Table[Random[Integer, {1, 2000}], {4000}];
frequencies[testlist]; // Timing
frequenciesAlt[testlist]; // Timing
```

```
{0.13, Null}

{0.01, Null}
```

```
testlist = Table[Random[Integer, {1, 20 000}], {15 000}];
frequencies[testlist]; // Timing
frequenciesAlt[testlist]; // Timing

{0.591, Null}
{0.07, Null}
```

We see that the version based on Sort and Split is several times faster. The reason that I have included the above more complex and less efficient implementation of < frequencies > is twofold. First, it is still a good illustration of how one can combine several programming techniques together (in this case, procedural (side-effects), functional and rule-based). Second, to show once again that it is very hard to outperform certain general functions such as Sort (this refers to "pure" Sort; Sort with a user-defined comparison function can be outperformed in some cases), and it is usually advantageous to use them if the problem can be reformulated in such a way that they can be used.

5.2.7 Apply

Apply is the second "fundamental" higher-order function in the FP programming paradigm. Its action is different from that of Map or related functions. Apply takes a function <f> as its first argument, and an expression <expr>as a second one. It changes the **head** of <expr> from what it was to <f>. Few simple examples:

5.2.7.1 Simple examples

```
Clear[a, b, c];
Apply[List, a + b + c]
{a, b, c}
```

To understand what has happened, recall the internal form of the sum above:

```
FullForm[a + b + c]
Plus[a, b, c]
```

So, the head Plus was changed to head List. Let us look at more examples like this:

```
Apply[Plus, a * b * c]
a + b + c
```

This time we changed head Times to head Plus. Now consider:

```
Range[10]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

This will give a sum of the first 10 natural numbers:

```
Apply[Plus, Range[10]]
55
```

And this will give 10!:

```
Apply[Times, Range[10]]
3 628 800
```

It is worth noting that this computation of the factorial is nearly as efficient as the built-in command:

```
Timing[Apply[Times, Range[10 000]];]
{0.05 Second, Null}
Timing[10 000 !;]
{0.04 Second, Null}
```

■ 5.2.7.2 Shorthand notation

As for other common operations, there is a shorthand notation for Apply: <**@@**>. So, to Apply a function <f> to expression <expr>, one uses

```
(f @@ expr)
```

Once again, parentheses can often be omitted, but are generally necessary to avoid precedence-related bugs.

By now we saw enough examples to understand in which cases Apply is really needed: these are the cases when some function needs the "interior" of some normal expression (i.e., comma-separated elements inside the square brackets, but not the head). So, what Apply does is that the present head of an expression gets "eaten up" by a new head, while all the comma-separated elements are "inherited" by a new head.

■ 5.2.7.3 More examples:

■ 5.2.7.3.1 Example: computing a quadratic norm of a tensor of arbitrary rank

This example we have already considered before (see section 3.8.3.3), but now we can fully understand it. This function computes the quadratic norm of the tensor of arbitrary rank:

```
Clear[tensorNorm];
tensorNorm[tensor_List] := Sqrt[Plus @@ Flatten[tensor^2]]
```

It works as follows: first, the list is squared (many built-in functions, and Power in particular, are Listable (section 4.9.1) and thus are automatically threaded over lists, so squaring a nested list of numbers is equivalent to squaring each number). Then, we use Flatten to make the list flat, by removing all the nested list structure (internal curly braces). Then we use Apply in the shorthand notation, to change the head from List to Plus. Finally, we take a square root of the resulting number.

For example, for a vector (list):

```
tensorNorm[Range[10]]
```

$\sqrt{385}$

for a 3x4 matrix:

```
(matrix = Table[i + j, {i, 1, 3}, {j, 1, 4}]) // MatrixForm
```

$$\begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

```
tensorNorm[matrix]
```

$\sqrt{266}$

- 5.2.7.3.2 Example: conditional summing of even numbers in a list

As a next example, consider the following problem: we have to write a function which sums a list of numbers, but it has to work only on a list with all numbers even. If this condition is not fulfilled, the function should return unevaluated.

To solve a problem, first create a sample list:

```
numlist = Range[10]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Summing it up is easy:

```
Plus @@ numlist
55
```

we can now define a function which sums up arbitrary list (for the sake of example, we will ignore the fact that the built-in Total does exactly that):

```
Clear[sumList];
sumList[x_List] := Plus @@ x;
```

Check:

```
sumList[{1, 2, 3, 4}]
10
```

Now we need to modify a function so that it checks our condition. Let us first work out the condition separately. The built-in function <EvenQ> checks whether the number is even. We have to do it for every number in our list, so we have to Map EvenQ on the list (again, for the sake of example we will ignore the fact that EvenQ gets automatically threaded over the list, and will do it manually). For our list:

```
Map[EvenQ, numlist]
{False, True, False, True, False, True, False, True, False, True}
```

All that is left is to plug this list into the built-in <And> command. But <And> receives not a list of expressions, but a sequence of (comma-separated) expressions, i.e, the interior of the list. Thus, the <List> head has to be "eaten up" by the <And> head, which means that we have to use Apply:

```
Apply[And, Map[EvenQ, numlist]]
False
```

or, which is the same,

```
And @@ Map[EvenQ, numlist]
False
```

The final step is to insert the condition into the pattern:

```
Clear[sumListEven];
sumListEven[x_List] /; And @@ Map[EvenQ, x] := Plus @@ x;
```

Check now:

```
sumListEven[{2, 4, 6, 8}]
20
sumListEven[{2, 4, 6, 7, 8}]
sumListEven[{2, 4, 6, 7, 8}]
```

Note the location of the condition pattern operator </;> - it is after the function parameter list, not inside. This is usually a better practice since for functions of more than one argument it allows to put conditions on several function parameters. Placing condition check inside a parameter list may force *Mathematica* to take global values for the parameters (instead of those passed to the function) to check the condition, which is probably not what you want. For more details on this, see [**See also [*****]**]

If we use the mentioned above Listable property of EvenQ (automatic threading over lists), the code will be somewhat more concise, and, more importantly, much faster :

```
Clear[sumList1];
sumList1[x_List] /; And @@ EvenQ[x] := Plus @@ x;
```

Let us now add one more definition. For instance, for all numbers odd we want to multiply them all:

```
sumList1[x_List] /; And @@ OddQ[x] := Times @@ x;
? sumList1
```

```
Global`sumList1

sumList1[x_List] /; And @@ EvenQ[x] := Plus @@ x

sumList1[x_List] /; And @@ OddQ[x] := Times @@ x
```

The rule has been added and the old one remained, although naively each definition contains the same pattern sumList1[x_List] . To resolve this paradox (see section 4.7.3), one has to realize that the condition check here is a part of the pattern. Therefore, the patterns for the two definitions really are different. This can also be seen very clearly with DownValues:

```
DownValues[sumList1]
{HoldPattern[sumList1[x_List] /; And @@ EvenQ[x]] :→ Plus @@ x,
 HoldPattern[sumList1[x_List] /; And @@ OddQ[x]] :→ Times @@ x}
```

A final word of caution: the detailed argument checks like the one in this problem, may induce a significant overhead in some cases. On the other hand, such checks are deceptively easy to write. In the present case, we were forced to do this because that's what is asked in the formulation of our model problem. If you are solving a large problem, then the design (splitting it into sub-problems/functions) is your decision. In this case, it is better not to supply each small function with condition checks like this (if possible), to avoid redundant checks (that is, in cases when the passed arguments are known beforehand to be fine). In terms of our "guiding principles for efficient programming", this refers to principle 4: "avoid complicated patterns".

```
Clear[numlist, sumList, sumList1, sumListEven];
```

- 5.2.7.3.3 Example: words containing given letters - realizing alternative patterns programmatically

For this example we again will need some list of words, like this one (taken from the *Mathematica* book)

```
wordlist = {"Most", "of", "this", "Part",
   "assumes", "no", "specific", "prior", "knowledge"
   , "of", "computer", "science", "Nevertheless", "some",
   "of", "it", "ventures", "into", "some", "fairly",
   "complicated", "issues", "You", "can", "probably", "ignore",
   "these", "issues", "unless", "they", "specifically",
   "affect", "programs", "you", "are", "writing"};
```

Now, the problem: we want to pick all the words containing any of the symbols given by some symbol list, for instance {"a","n","k"}.

To solve it, let us start with a simple case when we have only one symbol, say "a". And, as a first step, let us work out the code that will check for some single word (string), whether it contains the symbol. So, let us pick some word:

```
ourword = "fairly"
fairly
```

Since I don't want to use the string-matching functions here, the first thing we need is to split the word into its letters. This is best done by using the built-in <Characters > command:

```
Characters[ourword]
{f, a, i, r, l, y}
```

Now we need to test whether a given symbol ("a") belongs to this list. This is done best by the built-in <MemberQ> command:

```
MemberQ[Characters[ourword], "a"]
True
```

Now we want to generalize to the case of several characters. One way would be to Map MemberQ on their list, and then use the built-in <Or>. For instance, the second character is "k":

```
Map[MemberQ[Characters[ourword], #] &, {"a", "k"}]
{True, False}
```

We need to Apply <Or> now (head List has to be "eaten up" by Or)

```
Or @@ Map[MemberQ[Characters[ourword], #] &, {"a", "k"}]
True
```

We are now ready to insert this condition, and use Cases command to find all the words containing either "a" or "k" (or both)

```
Cases[wordlist,
 x_String /; Or @@ Map[MemberQ[Characters[x], #] &, {"a", "k"}]]
{Part, assumes, knowledge, fairly, complicated,
 can, probably, specifically, affect, programs, are}
```

Finally, we package everything into a function, which will take two arguments: a list of words and a list of symbols to look for:

```
Clear[findWordsWithSymbols];
findWordsWithSymbols[wlist_List, symbols_List] := Cases[wlist,
   x_String /; Or @@ Map[MemberQ[Characters[x], #] &, symbols]];
```

Note that we changed the specific symbol list by the function parameter <symbols>. To check it, let us find all words containing say "e" or "r" letters :

```
findWordsWithSymbols[wordlist, {"e", "r"}]
{Part, assumes, specific, prior, knowledge, computer,
 science, Nevertheless, some, ventures, some, fairly,
 complicated, issues, probably, ignore, these, issues,
 unless, they, specifically, affect, programs, are, writing}
```

We solved the problem, but rather inefficiently. Even within what we already know, there are ways to make it better. The main source of inefficiency which we can eliminate now is the Map-ping of < Member-Q[Characters[x],#]&> on a list of symbols. We can do better if we recall that the second argument of MemberQ is a pattern, and as such, it may be more complicated than just one symbol. In particular, we can use alternative patterns like "r"|"e", or, which is the same, Alternatives["r","e"].

```
MemberQ[Characters[ourword], Alternatives["a", "k"]]
True
```

Since again our letters are initially in a list, and Alternatives requires a sequence of elements, the head <List> has to be eaten up by the head "Alternatives", and therefore, we have to use Apply:

```
MemberQ[Characters[ourword], Apply[Alternatives, {"a", "k"}]]
True
```

or, which is the same,

```
MemberQ[Characters[ourword], Alternatives @@ {"a", "k"}]
True
```

we can now rewrite our function:

```
Clear[findWordsWithSymbolsAlt];
findWordsWithSymbolsAlt[wlist_List, symbols_List] := Cases[wlist,
   x_String /; MemberQ[Characters[x], Alternatives @@ symbols]];
```

Check:

```
findWordsWithSymbolsAlt[wordlist, {"n", "m"}]
```

```
{assumes, no, knowledge, computer, science, some, ventures, into,
 some, complicated, can, ignore, unless, programs, writing}
```

The reason that the latter version is more efficient than the former one is the following: in the latter case, the "decision" about any given word is made already on the level of the MemberQ function, while in the former case, it is promoted to another function (Or). The rule of thumb is that one has to push as much computation as possible inside the built-in function. Basically, MemberQ does not care (almost), whether it checks a single character or an alternative pattern with many characters (for small numbers of characters such as considered here). On the other hand, by Mapping MemberQ on each character, we force it to check afresh for every character. Thus, roughly we expect that the difference in performance will be a factor of the order of the length of the character list. We can check our expectations by measuring the timing for a list of symbols being the entire alphabet. We will use the <myTiming> function which measures small execution times (section 3.4.5.2). Now:

```
alphabet = Characters["abcdefghijklmnopqrstuvwxyz"]
```

```
{a, b, c, d, e, f, g, h, i, j, k,
 l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

```
myTiming[findWordsWithSymbols[wordlist, alphabet]]
```

```
0.014
```

```
myTiming[findWordsWithSymbolsAlt[wordlist, alphabet]]
```

```
0.000842
```

We see that we already gain more than an order of magnitude, for this example (the difference would be less for smaller list of symbols). As a byproduct, our new function is not only more efficient, but also more concise and transparent. In *Mathematica* programming this is very often the case. As one of our guiding principles has it, "Shorter programs usually run faster" (although, of course, there are exceptions).

Later we will see how to make this function yet more efficient, for example with the help of the Reap-Sow technique.

```
Clear[wordlist, findWordsWithSymbols,
  ourword, alphabet, findWordsWithSymbolsAlt];
```

5.2.7.3.4 Example: extracting matrix diagonals

Here we will be concerned with the following problem: given a square matrix, we need to extract all of its right diagonals (that is, diagonals going from top left to bottom right), and place them in a list. We will consider this problem in detail later in chapter VI, where 3 different solutions of it (in more general form) will be given. Now let us look at yet another one (not the most efficient), based on MapIndexed and using Apply in few places. In fact, as we will see, this is a good example to show many of the techniques discussed so far, and in particular how several built-in functions work together.

This is our test matrix :

```
testmatr = Array[Random[Integer, {1, 20}] &, {5, 5}];
testmatr // MatrixForm
```

$$\begin{pmatrix} 7 & 8 & 15 & 6 & 3 \\ 9 & 17 & 18 & 19 & 8 \\ 11 & 14 & 19 & 6 & 6 \\ 18 & 9 & 10 & 7 & 15 \\ 3 & 18 & 4 & 7 & 12 \end{pmatrix}$$

We first note that for any single right diagonal, the difference between its elements' indices (which map directly to the position indices in a nested list that we use to represent the matrix) is constant for all the elements. This means that we can "tag" matrix elements by these differences and then collect together those for which these "tags" will be the same. The only catch is that we have to ensure that the elements of the diagonals collected this way will be in right order. For our case, fortunately, this will be so due to the (depth - first) way how MapIndexed traverses the expressions.

So, let us start by "tagging" :

```
step1 = MapIndexed[{Subtract @@ #2, #1} &, testmatr, {2}]
```

```
{{{0, 7}, {-1, 8}, {-2, 15}, {-3, 6}, {-4, 3}},
 {{1, 9}, {0, 17}, {-1, 18}, {-2, 19}, {-3, 8}},
 {{2, 11}, {1, 14}, {0, 19}, {-1, 6}, {-2, 6}},
 {{3, 18}, {2, 9}, {1, 10}, {0, 7}, {-1, 15}},
 {{4, 3}, {3, 18}, {2, 4}, {1, 7}, {0, 12}}}
```

We used Apply to give to Subtract the interior of the position list for each element - that is, a sequence of vertical and horizontal positions rather than a list of them (which is the second argument that MapIndexed supplies to the function being mapped). We also see that we have to map on level {2}, since this is the level of individual matrix elements.

There are extra list brackets on level 1 which reflect the original separation of elements into rows, but which we don' t need any more. Thus, let us use Flatten on level 1 :

```
step2 = Flatten[step1, 1]
```

```
{{0, 7}, {-1, 8}, {-2, 15}, {-3, 6}, {-4, 3}, {1, 9},
 {0, 17}, {-1, 18}, {-2, 19}, {-3, 8}, {2, 11}, {1, 14},
 {0, 19}, {-1, 6}, {-2, 6}, {3, 18}, {2, 9}, {1, 10},
 {0, 7}, {-1, 15}, {4, 3}, {3, 18}, {2, 4}, {1, 7}, {0, 12}}
```

We will now sort this list with respect to the "tag", to make the elements of the same diagonal be adjacent to each other :

```
step3 = Sort[step2, First[#1] > First[#2] &]
{{4, 3}, {3, 18}, {3, 18}, {2, 4}, {2, 9}, {2, 11},
 {1, 7}, {1, 10}, {1, 14}, {1, 9}, {0, 12}, {0, 7}, {0, 19},
 {0, 17}, {0, 7}, {-1, 15}, {-1, 6}, {-1, 18}, {-1, 8},
 {-2, 6}, {-2, 19}, {-2, 15}, {-3, 8}, {-3, 6}, {-4, 3}}
```

Notice the use of Sort with a user-defined pure sorting criteria (in this case, sorting according to the "tag" which is a first element of each sublist).

This was actually a tricky step since here we are dependent on a particular way Sort works: there was no guarantee in principle that in the process of sorting it will not change the order of elements with the same value of the "tag". Fortunately for us, it works this way.

As a next step, we will use Split to group together the elements corresponding to the same diagonal :

```
step4 = Split[step3, First[#1] == First[#2] &]
{{{4, 3}}, {{3, 18}, {3, 18}},
 {{2, 4}, {2, 9}, {2, 11}}, {{1, 7}, {1, 10}, {1, 14}, {1, 9}},
 {{0, 12}, {0, 7}, {0, 19}, {0, 17}, {0, 7}},
 {{-1, 15}, {-1, 6}, {-1, 18}, {-1, 8}},
 {{-2, 6}, {-2, 19}, {-2, 15}}, {{-3, 8}, {-3, 6}}, {{-4, 3}}}
```

The next thing to do now is to extract the second element of each small sublist (which is the original matrix element) while preserving the structure of larger sublists which form the diagonals. This can be done by Map - ping the second element extraction on the level {2} of our nested list :

```
step5 = Map[#[[2]] &, step4, {2}]
{{3}, {18, 18}, {4, 9, 11}, {7, 10, 14, 9},
 {12, 7, 19, 17, 7}, {15, 6, 18, 8}, {6, 19, 15}, {8, 6}, {3}}
```

Another (and more efficient) way to do the same is to use Part with an extended functionality given by using the All specification :

```
step51 = Part[step4, All, All, 2]
{{3}, {18, 18}, {4, 9, 11}, {7, 10, 14, 9},
 {12, 7, 19, 17, 7}, {15, 6, 18, 8}, {6, 19, 15}, {8, 6}, {3}}
```

Since we are learning functional programming, I will keep the variant with Map in a final implementation, but again, the last one is more efficient and in principle should be used instead (if we were ultimately for efficiency, we should have chosen a different method in the first place - see chapter VI for details. Also, the use of Sort with a user-defined sorting function like above is not optimal).

The resulting lists are pretty much the diagonals, as you can see, but the elements in them are in reverse order (this is conventional. Our convention is that the diagonal starts at the top left corner). So, the last thing we have to do is to Map the Reverse function on our diagonal list :

```
result = Reverse /@ step5
```

```
{{3}, {18, 18}, {11, 9, 4}, {9, 14, 10, 7},
 {7, 17, 19, 7, 12}, {8, 18, 6, 15}, {15, 19, 6}, {6, 8}, {3}}
```

Now we combine everything into a function :

```
Clear[matrixRightDiags];
matrixRightDiags[matr_ ? MatrixQ] /; Equal @@ Dimensions[matr] :=
 Map[Reverse, Map[#[[2]] &, Split[
    Sort[Flatten[MapIndexed[{Subtract @@ #2, #1} &, matr, {2}], 1],
     First[#1] > First[#2] &], First[#1] == First[#2] &], {2}]]
```

Notice that we used the MatrixQ predicate to test that the input is a matrix, used a conditional pattern with the condition that the matrix is a square matrix, inside the condition used a built - in Dimensions which gives dimensions for a tensor, in a list, and used Apply another time to "eat up" the List head and give to Equal the sequence of matrix dimensions rather than a list.

So, we check once again :

```
matrixRightDiags[testmatr]
```

```
{{3}, {18, 18}, {11, 9, 4}, {9, 14, 10, 7},
 {7, 17, 19, 7, 12}, {8, 18, 6, 15}, {15, 19, 6}, {6, 8}, {3}}
```

This seems like a lot of work for something which can be in principle done with a doubly nested loop. But believe it or not, in practice, and with some experience, it is quite fast to write functions like this. Also, debugging is much easier than for a procedural version, because each line of code does a complete transformation and can be tested separately. Also (take it on faith for now, or have a look at chapter VI), for this particular problem the loop version will be terribly slow in *Mathematica*, even compared with the present one (not the most efficient). And in fact, if we think about it, MapIndexed used on level {2} represents exactly a nested loop, but done internally by *Mathematica*. Finally, this is a nice example of the interplay of different techniques (conditional patterns, mapping, level specification, Sort and Split with user - defined functions, pure functions) that we discussed in separation before.

As an exercise, you may consider the case of left diagonals (that is, those which start at bottom left and go to top right).

```
Clear[step1, step2, step3, step4, step5, step51, testmatr, result];
```

■ 5.2.7.4 Supplying a sequence of arguments to functions

There is one more very common use of Apply, which we will discuss now. This is when we have a partial list of arguments for some function stored in a separate list, and want to use that list in a function. As a simple example, let us define a function of five variables, which simply sums them up:

```
Clear[addFive, x, y, z, t, s, a, b];
addFive[x_, y_, z_, t_, s_] := x + y + z + t + s;
```

Suppose that we want to keep the first and the last arguments fixed, say at <a> and <b>. Now, say we have a list of 3-number lists:

```
testlist = Table[Random[Integer, {1, 10}], {10}, {3}]
{{1, 10, 6}, {9, 9, 2}, {9, 10, 10}, {7, 3, 6}, {2, 5, 7},
 {8, 6, 3}, {2, 3, 6}, {3, 8, 3}, {2, 6, 7}, {3, 10, 7}}
```

We would like to use our function on each sub-list, so that the numbers in the sublist will fill the three slots for the variables in the middle. In other words, we would like to Map our function on the list above, but in a way somewhat different from what we discussed before. As in the previous discussion (section 5.2.2.7), one solution would be to define an auxiliary function, which takes a list of three numbers:

```
Clear[g];
g[lst_List] /; Length[lst] == 3 :=
  addFive[a, lst[[1]], lst[[2]], lst[[3]], b];
```

Now we can Map <g>:

```
g /@ testlist
{17 + a + b, 20 + a + b, 29 + a + b, 16 + a + b, 14 + a + b,
 17 + a + b, 11 + a + b, 14 + a + b, 15 + a + b, 20 + a + b}
```

The problems with this solution are the same as with its analog that we discussed before. Before, we managed to find an alternative by using pure functions. Can we do the same here? To answer this, let us make our first attempt:

```
Map[addFive[a, #, b] &, testlist]
{addFive[a, {1, 10, 6}, b], addFive[a, {9, 9, 2}, b],
 addFive[a, {9, 10, 10}, b], addFive[a, {7, 3, 6}, b],
 addFive[a, {2, 5, 7}, b], addFive[a, {8, 6, 3}, b],
 addFive[a, {2, 3, 6}, b], addFive[a, {3, 8, 3}, b],
 addFive[a, {2, 6, 7}, b], addFive[a, {3, 10, 7}, b]}
```

We see now that we are almost there. The only stumbling block is the presence of a List head (curly braces) inside, which we would like to remove. We also know that Apply removes the head of an expression. But usually, it substitutes it by another (new) head, while here we would like none. It turns out that there exists a special head in *Mathematica*, which means exactly "no head". It is <Sequence>. So, our solution would be to Apply Sequence:

```
Map[addFive[a, Sequence @@ #, b] &, testlist]
{17 + a + b, 20 + a + b, 29 + a + b, 16 + a + b, 14 + a + b,
 17 + a + b, 11 + a + b, 14 + a + b, 15 + a + b, 20 + a + b}
```

Now it works and has the same advantages as the solution discussed before (section 5.2.2.7). I hasten to comment though that if one needs to Map function on a long list like here (or much longer still), sometimes there are better solutions available, like the one using built-in Thread (to be discussed below).

```
Clear[addFive, g, testlist];
```

5.2.7.5 Using Apply in conjunction with Map

Apply is often used in conjunction with Map. The typical situation is that we need the operation **Apply[function,expression]** to be Mapped on some list.

As an example, consider the following problem: we have a list of lists of numbers. The sublists have various length. We have to multiply all the numbers in each sublist. For example:

```
Clear[testlist];
testlist =
 Table[Random[Integer, {1, 10}], {10}, {Random[Integer, {2, 6}]}]
```

```
{{10, 3, 3, 6, 8}, {9, 3, 7, 7}, {10, 2, 10},
 {9, 3, 3, 5, 10, 3}, {7, 6, 2, 8, 5}, {1, 1, 2, 8, 2},
 {5, 1, 9, 10}, {8, 8, 7, 7, 8, 4}, {4, 3, 2, 2, 7}, {3, 8, 6}}
```

To multiply numbers in a single list, we use Apply[Times,list]. Now we have to Map it:

```
Map[Apply[Times, #] &, testlist]
```

```
{4320, 1323, 200, 12150, 3360, 32, 450, 100352, 336, 144}
```

The same can be done using the extended syntax for Apply, and supplying level as a third argument :

```
Apply[Times, testlist, 1]
```

```
{4320, 1323, 200, 12150, 3360, 32, 450, 100352, 336, 144}
```

This operation is so common that a special shorthand notation exists for it: Map[Apply[f,#]&,expr]== Apply[f,expr,1]==(f@@@expr). In our case:

```
Times @@@ testlist
```

```
{4320, 1323, 200, 12150, 3360, 32, 450, 100352, 336, 144}
```

Once again, one should be careful with the precedence, and generally the parentheses around the whole expression can not be dropped. For Mapping on the level(s) deeper than the first, there is no built-in shorthand notation, and one has to use Map with a proper level specification, or Apply with a third argument (again with a proper level specification).

```
Clear[testlist];
```

■ 5.2.8 When short-hands let us down: the Heads option

For all the functions described above, just as for functions such as Cases, Position etc. described in the previous chapter, there exists the Heads option. This option tells whether or not to make heads of expressions visible for these commands. Let me illustrate this on a few examples :

With Map :

```
Map[f, {{1, 2, 3}, {4, 5, 6}}]
Map[f, {{1, 2, 3}, {4, 5, 6}}, Heads → True]

{f[{1, 2, 3}], f[{4, 5, 6}]}

f[List][f[{1, 2, 3}], f[{4, 5, 6}]]
```

With MapIndexed

```
MapIndexed[f, {{1, 2, 3}, {4, 5, 6}}]
MapIndexed[f, {{1, 2, 3}, {4, 5, 6}}, Heads → True]

{f[{1, 2, 3}, {1}], f[{4, 5, 6}, {2}]}

f[List, {0}][f[{1, 2, 3}, {1}], f[{4, 5, 6}, {2}]]
```

With Scan

```
parts = {};
Scan[AppendTo[parts, #] &, {{1}, {2}}, Infinity];
parts

{1, {1}, 2, {2}}

parts = {};
Scan[AppendTo[parts, #] &, {{1}, {2}}, Infinity, Heads → True];
parts

{List, List, 1, {1}, List, 2, {2}}
```

What these examples illustrate is that setting Heads -> True makes the heads of (sub) expressions visible to Map, MapIndexed, Scan, MapAll or Apply (the latter with explicit levspec given).

Note that for the above function, the default is Heads -> False. It is not possible to set this option when shorthands like /@, @@ , @@@ are used.

If you will be the only user of a particular program you are writing, it is perhaps less important to keep track of this option settings since you can always correct things yourself. Besides, since the default is Heads -> False, which in the overwhelming majority of cases is what is needed, there seems nothing to worry about. However, if your program will be used by someone else, it is essential to indicate the Heads -> False option explicitly (even though this is the default) every time that you use one of the above commands. The point is that if you don’t, and the person who uses your function (perhaps, yourself a few months later!) has set Heads -> True globally by SetOptions command (which is a bad practice by the way), for Map or Apply etc, then your program will use that option instead, and consequently will probably not work correctly. This issue is especially important when writing packages - the custom extensions of *Mathematica* to some domain. And for this reason, it is best to avoid short - hand notation for Map, Apply etc in the final version of the code, since you can not set this option in the short - hand notation.

5.3 Generalizations

5.3.1 Thread

This function threads a function of several variables over the list in which first sublist gives all first arguments, second gives second arguments, etc. For example:

5.3.1.1 Initial examples

```
Clear[f];
Thread[f[Range[10], Range[11, 20]]]
{f[1, 11], f[2, 12], f[3, 13], f[4, 14], f[5, 15],
 f[6, 16], f[7, 17], f[8, 18], f[9, 19], f[10, 20]}
Thread[f[Range[10], Range[11, 20], Range[21, 30]]]
{f[1, 11, 21], f[2, 12, 22], f[3, 13, 23], f[4, 14, 24], f[5, 15, 25],
 f[6, 16, 26], f[7, 17, 27], f[8, 18, 28], f[9, 19, 29], f[10, 20, 30]}
```

The lists of arguments need not be the same length, and this is in fact quite useful at times:

```
Thread[f[Range[10], 1]]
{f[1, 1], f[2, 1], f[3, 1], f[4, 1], f[5, 1],
 f[6, 1], f[7, 1], f[8, 1], f[9, 1], f[10, 1]}
```

When used in cases like above, Thread may be thought of as a generalization of Map.

However, the input like this is ambiguous, and the system complains:

```
Thread[f[Range[10], {1, 2}]]
Thread::tdlen : Objects of unequal length in
   f[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {1, 2}] cannot be combined. More…
f[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {1, 2}]
```

5.3.1.2 Example: imitating Thread

As an amusing exercise, we may imitate the workings of Thread in the first case above. This will also clarify how it works in that case.

This will imitate Thread for the inputs with equal number of all arguments

```
Clear[myThread];
myThread[f_[x__List]] /; Equal @@ Map[Length, {x}] :=
  f @@@ Transpose[{x}];
myThread[x_List] := x;
```

In the above, the pattern checks that all elements are lists, and the condition checks that all lengths of all lists are the same (the second definition is needed to reproduce the behavior of the built-in Thread on simple lists). We then Transpose the lists, for instance:

```
Transpose[{Range[10], Range[11, 20], Range[21, 30]}]
{{1, 11, 21}, {2, 12, 22}, {3, 13, 23}, {4, 14, 24}, {5, 15, 25},
 {6, 16, 26}, {7, 17, 27}, {8, 18, 28}, {9, 19, 29}, {10, 20, 30}}
```

and we want to Map f on the resulting list of lists containing arguments. Since the List head in each sublist has to be substituted by the head <f>, we actually Map the Apply[f,#]&, and then use the shorthand notation discussed before (see section 5.2.7.5).

```
myThread[f[Range[10], Range[11, 20], Range[21, 30]]]
{f[1, 11, 21], f[2, 12, 22], f[3, 13, 23], f[4, 14, 24], f[5, 15, 25],
 f[6, 16, 26], f[7, 17, 27], f[8, 18, 28], f[9, 19, 29], f[10, 20, 30]}
```

It is left as an exercise for the reader to imitate the behavior of Thread in other cases (like the one where some of the arguments are not lists but atoms, like in the second example above).

Of course, we expect our function to be much slower than the built-in. Let us see how much slower:

```
myThread[f[Range[100], Range[101, 200], Range[201, 300]]] // myTiming
0.000211
Thread[f[Range[100], Range[101, 200], Range[201, 300]]] // myTiming
0.00015
```

In this case, the difference is about 50-100%, which should mean that we did not a bad job (however keep in mind that we did not cover more complicated uses of Thread in our function - this is likely to increase the gap in performance)

- 5.3.1.3 Performance study: redoing the Mapping-a-function-with-several-arguments example with Thread

Let us return to the example

```
Thread[f[Range[10], 1]]
{f[1, 1], f[2, 1], f[3, 1], f[4, 1], f[5, 1],
 f[6, 1], f[7, 1], f[8, 1], f[9, 1], f[10, 1]}
```

It shows that one of the cases when Thread is particularly useful is when one needs to supply a function with some arguments which are the same (don't change). We already discussed how to do this using Map and Apply, and here is an alternative. We may now redo our previous examples using Thread:

In our first example (c.f. section 5.2.2.7) we have a function:

```
Clear[f, a];
f[x_, y_] := Sin[x + y];
```

And we want to Map it on a list {1,2,3,4,5}, with the variable <y> fixed at value <a>. This is how we would do it with Thread:

```
Thread[f[Range[5], a]]
{Sin[1 + a], Sin[2 + a], Sin[3 + a], Sin[4 + a], Sin[5 + a]}
```

Since this is a more direct use of the built-in commands, we should expect it to be more efficient than the previous one with Map.

```
Map[f[#, a] &, Range[5]]
{Sin[1 + a], Sin[2 + a], Sin[3 + a], Sin[4 + a], Sin[5 + a]}
```

We may now verify this:

```
Thread[f[Range[50], a]] // myTiming
0.00024
```

```
Map[f[#, a] &, Range[50]] // myTiming
0.000331
```

We see that we gain about 30-40% here, even though the method with Map is by far not the worst.

- 5.3.1.4 Performance study: redoing a supplying-function-arguments example with Thread

Let us also redo the second example:

```
Clear[addFive, x, y, z, t, s, a, b];
addFive[x_, y_, z_, t_, s_] := x + y + z + t + s;
```

We want to keep the first and the last arguments fixed, say at <a> and <b>. Here is the list of other arguments

```
testlist = Table[Random[Integer, {1, 10}], {10}, {3}]
{{10, 10, 2}, {5, 6, 6}, {8, 7, 3}, {10, 8, 10}, {2, 5, 5},
 {1, 2, 8}, {5, 8, 5}, {3, 4, 5}, {3, 10, 6}, {7, 2, 4}}
```

Version with map and Apply:

```
Map[addFive[a, Sequence @@ #, b] &, testlist]
{22 + a + b, 17 + a + b, 18 + a + b, 28 + a + b, 12 + a + b,
 11 + a + b, 18 + a + b, 12 + a + b, 19 + a + b, 13 + a + b}
```

For the version with Thread:

```
Thread[addFive[a, Sequence @@ Transpose[testlist], b]]
{22 + a + b, 17 + a + b, 18 + a + b, 28 + a + b, 12 + a + b,
 11 + a + b, 18 + a + b, 12 + a + b, 19 + a + b, 13 + a + b}
```

To be able to use Thread, we had here to Transpose the list and then Apply Sequence, since the list was already in the form where all arguments for each function application are grouped together, while Thread normally works with them stored in a separate lists. We can now check the performance:

```
testlist = Table[Random[Integer, {1, 10}], {50}, {3}];
```

```
Map[addFive[a, Sequence @@ #, b] &, testlist] // myTiming
0.000871
Thread[addFive[a, Sequence @@ Transpose[testlist], b]] // myTiming
0.000231
```

Notice that the difference is about 4 times, even given an additional command had to be executed inside the Thread! So, here we gain an increase in performance by several times. Once again, this teaches us that in *Mathematica* programming it is important to choose the right idiom. Here, for instance, the length of the code is comparable in both cases, but the solution using Thread picked a better idiom for this particular problem. In order to understand this behavior, we should realize that in Thread, the parameters <a> and <b> are treated internally by Thread. Also, the substitution of the middle arguments to f is performed case-by-case in the approach with Map, while done internally for all elements in the list by Thread. It basically does the same thing that we do with the Map command, but does more of it internally, and thus does it faster.

In this respect the *Mathematica* language is more like the natural language than one of the more standard programming languages. Like in the natural language, there are plenty of ways to express the same thing. And like in the natural language, using the most precise idiom is advantageous. It takes some time to develop this skill but it pays off - after all, there are not so many fundamental commands in *Mathematica*. Once you get to know how to use them, the rest will follow.

```
Clear[addFive, testlist];
```

■ 5.3.1.5 Example: simple encoding - using Thread to create a list of rules

One particular case when Thread is quite useful is when we have to create a set of rules. In this example we will build a function which does simple encodings, by substituting each letter in a message by some another letter.

To start, let us create an alphabet list:

```
alphabet = Characters["abcdefghijklmnopqrstuvwxyz"]
{a, b, c, d, e, f, g, h, i, j, k,
 l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

Now, let us shift all the letters by, say,10:

```
shifted = RotateRight[alphabet, 10]
{q, r, s, t, u, v, w, x, y, z, a,
 b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}
```

Now we will create the encoding rules:

```
coderules = Thread[Rule[alphabet, shifted]]
{a → q, b → r, c → s, d → t, e → u, f → v, g → w, h → x,
 i → y, j → z, k → a, l → b, m → c, n → d, o → e, p → f, q → g,
 r → h, s → i, t → j, u → k, v → l, w → m, x → n, y → o, z → p}
```

What happened was that the function Rule was threaded over the lists, producing a list of rules, in full agreement with the general way the Thread works. It may help to look at the FullForm here:

```
FullForm[coderules]
```

```
List[Rule["a", "q"], Rule["b", "r"], Rule["c", "s"], Rule["d", "t"],
 Rule["e", "u"], Rule["f", "v"], Rule["g", "w"], Rule["h", "x"],
 Rule["i", "y"], Rule["j", "z"], Rule["k", "a"], Rule["l", "b"],
 Rule["m", "c"], Rule["n", "d"], Rule["o", "e"], Rule["p", "f"],
 Rule["q", "g"], Rule["r", "h"], Rule["s", "i"], Rule["t", "j"], Rule["u", "k"],
 Rule["v", "l"], Rule["w", "m"], Rule["x", "n"], Rule["y", "o"], Rule["z", "p"]]
```

To proceed, consider some test message, like

```
message = "never say never"
```

```
never say never
```

To apply the rules, we have to break the message into characters:

```
Characters[message]
```

```
{n, e, v, e, r,  , s, a, y,  , n, e, v, e, r}
```

We now apply the rules:

```
Characters[message] /. coderules
```

```
{d, u, l, u, h,  , i, q, o,  , d, u, l, u, h}
```

Finally, we have to assemble the encoded message back. For this, we will use the **StringJoin** built-in function, and since the head List has to be eaten up, we use Apply:

```
Apply[StringJoin, Characters[message] /. coderules]
```

```
duluh iqo duluh
```

Now we can package these steps into an encoding function:

```
Clear[encode];
encode[mes_String, rules_List] :=
  Apply[StringJoin, Characters[mes] /. rules];
```

Check:

```
coded = encode[message, coderules]
```

```
duluh iqo duluh
```

To decode the message back, we don't need another function. All we need to do is to reverse the rules. These are the rules:

```
coderules
```

```
{a → q, b → r, c → s, d → t, e → u, f → v, g → w, h → x,
 i → y, j → z, k → a, l → b, m → c, n → d, o → e, p → f, q → g,
 r → h, s → i, t → j, u → k, v → l, w → m, x → n, y → o, z → p}
```

If you look now at the FullForm (above), it is clear that we have to reverse the order of letters inside each Rule. There is a built-in function Reverse. Let us check on a single Rule that it will work:

```
Clear[a];
Reverse["a" → "q"]
q → a
```

All we have to do now is to Map it on our list of rules:

```
revrules = Reverse /@ coderules
{q → a, r → b, s → c, t → d, u → e, v → f, w → g, x → h,
 y → i, z → j, a → k, b → l, c → m, d → n, e → o, f → p, g → q,
 h → r, i → s, j → t, k → u, l → v, m → w, n → x, o → y, p → z}
```

Now we can decode the message back:

```
decoded = encode[coded, revrules]
never say never
```

This is a simple example where you can see a nice coexistence and complementarity of rule-based and functional programming styles.

```
Clear[alphabet, coderules,
  encode, revrules, message, decoded, coded];
```

- 5.3.1.6 Example: unsorted union problem revisited

One of the past examples for the <MapIndexed> function was to create an unsorted union of elements for some list. In particular, a set of rules for elements to their positions was constructed using MapIndexed. The code for the <unsortedUnion> function looked like:

```
Clear[unsortedUnion];
unsortedUnion[x_List] :=
  Extract[x, Sort[Union[x] /. Dispatch[MapIndexed[Rule, x]]]];
```

The same rules can be created using Thread, which we will do now. This will be our test list:

```
testlist = Table[Random[Integer, {1, 10}], {20}]
{2, 3, 7, 7, 9, 10, 10, 1, 1, 5, 8, 10, 5, 8, 6, 3, 5, 4, 4, 3}
```

This will create a set of rules similar to the one we have previously created by MapIndexed:

```
Thread[Rule[testlist, Range[Length[testlist]]]]
{2 → 1, 3 → 2, 7 → 3, 7 → 4, 9 → 5, 10 → 6,
 10 → 7, 1 → 8, 1 → 9, 5 → 10, 8 → 11, 10 → 12, 5 → 13,
 8 → 14, 6 → 15, 3 → 16, 5 → 17, 4 → 18, 4 → 19, 3 → 20}
```

The only difference is that the positions are not wrapped in curly braces (lists), as they were for MapIndexed:

```
MapIndexed[Rule, testlist]
```

```
{2 → {1}, 3 → {2}, 7 → {3}, 7 → {4}, 9 → {5}, 10 → {6}, 10 → {7},
 1 → {8}, 1 → {9}, 5 → {10}, 8 → {11}, 10 → {12}, 5 → {13}, 8 → {14},
 6 → {15}, 3 → {16}, 5 → {17}, 4 → {18}, 4 → {19}, 3 → {20}}
```

We could Map List on the result in our Thread-based realization, like this:

```
Thread[Rule[testlist, List /@ Range[Length[testlist]]]]
```

```
{2 → {1}, 3 → {2}, 7 → {3}, 7 → {4}, 9 → {5}, 10 → {6}, 10 → {7},
 1 → {8}, 1 → {9}, 5 → {10}, 8 → {11}, 10 → {12}, 5 → {13}, 8 → {14},
 6 → {15}, 3 → {16}, 5 → {17}, 4 → {18}, 4 → {19}, 3 → {20}}
```

However, there is a more efficient realization - to keep the list as it is, but use Part instead of Extract (Part has a somewhat different syntax and in particular accepts a simple list of positions like the one generated by Thread). So, this is the final version:

```
Clear[unsortedUnionNew];
unsortedUnionNew[x_List] := Part[x,
   Sort[Union[x] /. Dispatch[Thread[Rule[x, Range[Length[x]]]]]]];
```

Check:

```
unsortedUnionNew[testlist]
```

```
{2, 3, 7, 9, 10, 1, 5, 8, 6, 4}
```

We can now compare performance on some large list:

```
testlist = Table[Random[Integer, {1, 1000}], {4000}];
```

```
unsortedUnion[testlist] // myTiming
```

```
0.0261
```

```
unsortedUnionNew[testlist] // myTiming
```

```
0.025
```

The performance is roughly the same.

```
Clear[testlist, unsortedUnion, unsortedUnionNew];
```

There are more capabilities of the Thread command. Some of them are discussed in *Mathematica* Help. We will eventually discuss them as we get to examples where they are useful.

■ 5.3.2 MapThread

MapThread is a close cousin of Thread, with several important differences. First, the format of the command is somewhat different:

```
MapThread[function, {arglist1, arglist2, ...}]
```

For MapThread, unlike Thread, the lists of arguments <arglist1,...> should all be of the same length.

- 5.3.2.1 Simple examples

Threading

This does the same as Thread, for a generic head (function) <f>:

```
Clear[f];
MapThread[f, {Range[10], Range[11, 20]}]
```

```
{f[1, 11], f[2, 12], f[3, 13], f[4, 14], f[5, 15],
 f[6, 16], f[7, 17], f[8, 18], f[9, 19], f[10, 20]}
```

Multiplying numbers pairwise in two lists

Here we use a concrete function Times to multiply the numbers in the two lists pairwise.

```
MapThread[Times, {Range[10], Range[11, 20]}]
```

```
{11, 24, 39, 56, 75, 96, 119, 144, 171, 200}
```

Notice that this operation is done easier by just multiplying two lists (this is possible because Times is a Listable operation):

```
Range[10] * Range[11, 20]
```

```
{11, 24, 39, 56, 75, 96, 119, 144, 171, 200}
```

As it became a habit, let us digress to measure relative performance:

```
MapThread[Times, {Range[100], Range[101, 200]}] // myTiming
```

```
0.000211
```

```
Range[100] * Range[101, 200] // myTiming
```

```
0.000017
```

Here we find a 15-20 times difference! In fact, by trying smaller and larger lists you can convince yourself that this factor is not constant. Instead, the performance gap increases even more as the lists get longer. The reason is that the operations like list multiplication (also dot product etc) are highly optimized in *Mathematica*, while MapThread is a good, but general purpose command.

For the record, in this particular case, and for machine - size numbers, a cheap way to speed - up MapThread is to compile the code:

```
(comp = Compile[{{x, _Integer, 1}, {y, _Integer, 1}},
    MapThread[Times, {x, y}]]) // myTiming
```

```
0.000311
```

```
comp[Range[100], Range[101, 200]] // myTiming
```

```
0.0000541
```

As is clear from these timings, this will pay off (as compared to the uncompiled version) if many operations such as this are needed, since compilation also takes some time. And also, even the compiled version is about 3 times slower than the one based on Times being Listable.

5.3.2.2 Thread and MapThread: important difference in evaluation

There is one more important difference between Thread and MapThread which I would like to illustrate now. For this purpose, let us measure also the performance of Thread on the same problem:

```
Thread[Times[Range[100], Range[101, 200]]] // myTiming
0.0000281
```

It looks like Thread performs here an order of magnitude faster than MapThread. But this is an illusion.To see what really happens, let us Trace the execution for small lists:

```
Thread[Times[Range[10], Range[11, 20]]] // Trace
{{{Range[10], {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}},
  {Range[11, 20], {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}},
  {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
  {11, 24, 39, 56, 75, 96, 119, 144, 171, 200}},
 Thread[{11, 24, 39, 56, 75, 96, 119, 144, 171, 200}],
 {11, 24, 39, 56, 75, 96, 119, 144, 171, 200}}
```

What we see is that the Times command is evaluated, producing the final list, before Thread has any chance to execute. Thus, the role of Thread here is just to stay idle. This is why the performance is so close to the one given by direct multiplication - the main work is again done by the Times command. In fact, what happened was to be expected: the standard evaluation procedure consists in evaluating inner expressions before the outer ones. So, Thread evaluates its arguments in the standard way.

While MapThread also evaluates arguments in a standard way, it has a different syntax where the function to be threaded is a separate argument of MapThread. Thus, the above behavior can not happen here. This constitutes one important difference between Thread and MapThread functionality.

■ 5.3.2.3 Case study: checking lists for equality

Checking lists for equality

One could use MapThread for element-by-element comparison of several lists:

```
MapThread[Equal, {{1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}}]
{True, True, True, True, True}
```

Now,

```
MapThread[Equal, {{1, 2, 10, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}}]
{True, True, False, True, True}
```

We need to Apply And to get a final result:

```
And @@
 MapThread[Equal, {{1, 2, 10, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}}]
False
```

Making a listEqualQ function

We can now make a function:

```
Clear[listsEqualQ];
listsEqualQ[lists__List] /; Equal @@ Map[Length, {lists}] :=
  And @@ MapThread[Equal, {lists}];
listsEqualQ[lists__List] := False;
```

I have used this opportunity to illustrate several issues. First, the pattern <lists__List> immediately ensures that the function is defined on any non-zero number of lists, and only lists - otherwise the pattern will not match. Next, we attach a condition that all lists are of equal length. If this is not so, the conditional pattern in the first line will not match, but the pattern in the second definition is more general and will match - we will get False then, since we consider the lists of different lengths to be always different. At the same time, this condition-checking ensures that MapThread on the r.h.s will always receive lists of equal length, which is a pre-requisite for MapThread.

Check:

```
listsEqualQ[{1, 2, 3}, {1, 2, 3}]
True
listsEqualQ[{1, 2, 3}, {1, 2, 4}]
False
listsEqualQ[{1, 2, 3}, {1, 2}]
False
listsEqualQ[{1, 2, 3}, a, {1, 2, 3}]
listsEqualQ[{1, 2, 3}, a, {1, 2, 3}]
```

Notice the last case: the function remained unevaluated rather than giving False. This behavior is consistent with the general *Mathematica* ideology that whenever the system can not decide, the expression should return unevaluated. In particular, we may consider the following code:

```
Clear[a];
answer = listsEqualQ[{1, 2, 3}, a, {1, 2, 3}];
a = {1, 2, 3};
answer

True
```

At the time when <answer> was computed, the value of <a> was such that there was no definite result (<a> had no value). Later <a> received a value, which enabled <answer> to evaluate do a definite value (True in this case). I don't want to encourage this style of programming (dependence on global variables in this fashion), but just to illustrate that our function <listsEqualQ> has a standard behavior, expected normally from *Mathematica* built-in functions. Had we used as a second definition something like <listsEqualQ[x_]:=False>, this would produce False on any input not matching the pattern of the first definition, and the behavior would be different. In general, it is a good practice to try make your own functions behave as much as built-in ones, as possible.

Performance analysis

Let us look now at the performance of our function. An immediate comment here is that the built-in Equal works on lists, which means that our function will almost certainly be slower or much slower. Let us see how much slower:

```
listsEqualQ[Range[1000], Range[1000], Range[1000]] // myTiming
0.0031
Equal[Range[1000], Range[1000], Range[1000]] // myTiming
0.0000301
```

I get about 100 times difference on my machine, for the length of the lists equal 1000. In fact, this coefficient is not constant but depends on the size of the lists and will increase with it - here we have different computational complexities. Let us see how much faster we can go if we drop the equal-length condition and pattern-matching:

```
And @@ MapThread[Equal, {Range[1000], Range[1000], Range[1000]}] //
 myTiming
0.0019
```

We see that we get about 1.5 increase in performance by doing so, but are still miles away from the built-in function Equal. In the particular case of the present problem, there exists another way of doing this with performance roughly equivalent to our previous implementation:

```
Apply[And, Equal @@@ Transpose[{Range[10], Range[10], Range[10]}]]
True
Apply[And, Equal @@@
   Transpose[{Range[1000], Range[1000], Range[1000]}]] // myTiming
0.003
```

Do you understand the way the code works in this case?

A faster implementation

To complete this story, let me display a solution which is more tricky, still much slower than the built-in Equal, but can give a factor of 4-5 increase in performance as compared to the above (in fact, the difference is more in *Mathematica* 5.. versions than in *Mathematica* 6, where the code below seems to work about twice slower for some reason):

```
Clear[listsEqualQNew];
listsEqualQNew[lists__List] /; Equal @@ Map[Length, {lists}] :=
  Plus @@ Flatten[Abs[
      Apply[Subtract, {lists}[[#]]] & /@
       Partition[Range[Length[{lists}]], 2, 1]]] === 0;
listsEqualQNew[lists__List] := False;
```

The idea is that we pairwise subtract the lists, using a high-performance Subtract operation which also works on lists of the same length. Then we take an absolute values of the results and sum them all. The result has to be zero if all lists are equal, otherwise it will be non-zero. To account for symbolic expressions, the <SameQ> (===) operator is used. <Partition > operator is used to create a list of pairs of positions, and <{lists}[[#]]&> function extracts from the list {lists} the pair of lists corresponding to those positions. It is a good exercise to take some small lists and dissect this function, to understand each step. We check now:

```
listsEqualQNew[{1, 2, 3}, {1, 2, 3}, {1, 2, 3}]
True
listsEqualQNew[{1, 2, 3}, {1, 2, 4}, {1, 2, 3}]
False
listsEqualQNew[Range[1000], Range[1000], Range[1000]] // myTiming
0.0012
Equal[Range[1000], Range[1000], Range[1000]] // myTiming
0.000033
```

A tricky point, and more on attributes

Notice however, that there is one instance in which the latter implementation will not work correctly : when the tested lists contain sublists of different lengths (well, it sort of works, but generates error messages):

```
listsEqualQNew[{{1, 2}, {3, 4}}, {{1, 2, 3}, {1}}]
Thread::tdlen : Objects of unequal length in {1, 2} – {1, 2, 3} cannot be combined. ≫
Thread::tdlen : Objects of unequal length in {3, 4} – {1} cannot be combined. ≫
False
```

This is because the addition and subtraction is defined on lists, but of the same length (this is what Listable attribute does). We can get rid of this by temporarily removing the Listable attribute for Subtract function - this is the first tricky point. The second is that we also have to do the same for Plus function (this may not be obvious, but Subtract is really more like a wrapper, the real work being done by Plus). Our new function will look like:

```
Clear[listsEqualQNew1];
listsEqualQNew1[lists__List] /; Equal @@ Map[Length, {lists}] :=
  Module[{result},
   ClearAttributes[{Plus, Subtract}, Listable];
   result = (Plus @@ Flatten[Abs[
         Apply[Subtract, {lists}[[#]]] & /@
          Partition[Range[Length[{lists}]], 2, 1]]] === 0);
   AppendTo[Attributes[Plus], Listable];
   AppendTo[Attributes[Subtract], Listable];
   Return[result]];

listsEqualQNew[lists__List] := False;
```

Notice that we remove the attributes first, with the help of another useful command: **ClearAttributes**. Then we compute the function result proper, and then restore the attributes. Notice that we did not use the SetAttributes function to change attributes in this example. In fact, if we try to use SetAttributes on Plus, it does not work:

```
SetAttributes[Plus, DeleteCases[Attributes[Plus], Listable]];
Attributes[Plus]
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

We see that when we attempt to do it in this way, Listable attribute remains.

Note also that neither did we Unprotect Plus. The idiom Attributes[command] = {attribute list} is then rather dangerous because one can easily change the behavior of the built - in functions with it, and no warning message will be generated. The protection of functions by Protected attribute protects the function symbol, but not the function attributes.

Anyway, let us check our function :

```
listsEqualQNew1[{{1, 2}, {3, 4}}, {{1, 2, 3}, {1}}]
False
```

It works fine now. The above modification leads to a slight decrease in performance however:

```
listsEqualQNew1[Range[1000], Range[1000], Range[1000]] // myTiming
0.0017
```

Again, for some reason this function works 2 - 3 times faster in version 5.2 (where it is then 5-6 times faster than our previous less sophisticated implementation), than in version 6.

We see that our best implementation is still way slower than a built-in. Note however that in cases where you need not only a final answer about equality, but for example the information about which elements in the list break that equality, you will need something like what we implemented above, since the built-in Equal does not give you such details.

■ 5.3.2.4 More examples

■ 5.3.2.4.1 Example: replacing the main diagonal in the square matrix

Consider the following problem: we are given a square matrix of dimension <n> (say,5), and a list of the same length. We want to replace the main diagonal of the matrix by this list.

Say, this is our matrix:

```
(matrix = Table[Random[Integer, {1, 10}], {5}, {5}]) // MatrixForm
```

$$\begin{pmatrix} 10 & 3 & 6 & 9 & 4 \\ 4 & 8 & 2 & 2 & 6 \\ 5 & 9 & 5 & 5 & 2 \\ 4 & 3 & 6 & 9 & 3 \\ 5 & 7 & 2 & 3 & 7 \end{pmatrix}$$

And this is our replacement list:

```
Clear[a, b, c, d, e];
replist = {a, b, c, d, e}
```

```
{a, b, c, d, e}
```

To solve the problem, we will use the built-in function ReplacePart, in the form in which it takes 3 arguments: the expression, the new value of the element, and its position in the expression. For instance,

```
ReplacePart[{1, 2, 3, 4, 5}, a, 3]
```

```
{1, 2, a, 4, 5}
```

Then, this is the code to solve our problem:

```
result = MapThread[ReplacePart, {matrix, replist, Range[5]}]
```

```
{{a, 3, 6, 9, 4}, {4, b, 2, 2, 6},
 {5, 9, c, 5, 2}, {4, 3, 6, d, 3}, {5, 7, 2, 3, e}}
```

Be sure to understand how this code works. To display the result in the form of the matrix, we use MatrixForm:

```
MatrixForm[result]
```

$$\begin{pmatrix} a & 3 & 6 & 9 & 4 \\ 4 & b & 2 & 2 & 6 \\ 5 & 9 & c & 5 & 2 \\ 4 & 3 & 6 & d & 3 \\ 5 & 7 & 2 & 3 & e \end{pmatrix}$$

The other way to perform the same operation is using MapIndexed:

```
MapIndexed[If[Equal @@ #2, replist[[#2[[1]]]], #1] &, matrix, {2}]
```

```
{{a, 3, 6, 9, 4}, {4, b, 2, 2, 6},
 {5, 9, c, 5, 2}, {4, 3, 6, d, 3}, {5, 7, 2, 3, e}}
```

What this does it to Map on every element of the matrix a function which changes the element to a corresponding element of the replacement list, if the element is on the diagonal, and returns the element back if it is not. However, here we know a priori that this implementation will be inefficient compared to the previous one, since its complexity is quadratic with the matrix size while it is linear for the previous one (in the former case, it sweeps through all matrix elements, not just the diagonal ones).

Finally we package our solution into a function:

```
Clear[replaceDiagonal];
replaceDiagonal[matrix_ ? MatrixQ, replist_List] /; Equal[
    Length[replist], Sequence @@ Dimensions[matrix]] := MapThread[
   ReplacePart, {matrix, replist, Range[Length[replist]]}];
```

I used this opportunity to introduce another two built-in functions: <**Dimensions**>, which gives a list of dimensions of a nested list (a matrix in this case), and the predicate <**MatrixQ**> which determines whether or not an object is a matrix. I also used the idiom Apply[Sequence,expression] once again (section 5.2.7.5).

Basically, the attached condition checks that both matrix dimensions are the same and equal to the length of the replacement list. Let us check now:

```
replaceDiagonal[matrix, Range[5]^3] // MatrixForm
```

$$\begin{pmatrix} 1 & 3 & 6 & 9 & 4 \\ 4 & 8 & 2 & 2 & 6 \\ 5 & 9 & 27 & 5 & 2 \\ 4 & 3 & 6 & 64 & 3 \\ 5 & 7 & 2 & 3 & 125 \end{pmatrix}$$

```
replaceDiagonal[{{1, 2}, {3, 4}}, {0, 0}] // MatrixForm
```

$$\begin{pmatrix} 0 & 2 \\ 3 & 0 \end{pmatrix}$$

```
replaceDiagonal[{{1, 2}, {3, 4}}, {0, 0, 1}]
replaceDiagonal[{{1, 2}, {3, 4}}, {0, 0, 1}]
```

```
replaceDiagonal[{{1, 2, 5}, {3, 4, 6}}, {0, 0}]
replaceDiagonal[{{1, 2, 5}, {3, 4, 6}}, {0, 0}]
```

It is left as an exercise to the reader to package an alternative solution with MapIndexed into another function, test it and then study the relative performance of the two functions on matrices of various sizes, to confirm our expectations of linear vs quadratic complexity of the two solutions.

```
Clear[replaceDiagonal, matrix, replist, result];
```

- 5.3.2.4.2 Example: appending sublists of a nested list

Here we are concerned with the following problem. Given a nested list of numbers (with sublists of generally different length), and a list of separate list of numbers of the same length as the nested list, append each number of the simple list to the end of the corresponding sublist of the nested list.

This is our nested list

```
Clear[testlist];
testlist =
 Table[Random[Integer, {1, 10}], {10}, {Random[Integer, {2, 6}]}]
```

```
{{10, 7}, {9, 7, 2}, {3, 9, 8, 2, 3}, {4, 6, 6, 4}, {9, 2, 5, 6, 3, 3},
 {1, 1}, {1, 2, 5}, {3, 10, 3, 1, 5, 5}, {8, 10}, {6, 10, 7}}
```

This is our simple list

```
addlist = Table[Random[Integer, {1, 10}], {10}]
{4, 10, 6, 6, 1, 8, 5, 7, 1, 6}
```

This is the code that solves the problem

```
result = MapThread[Append, {testlist, addlist}]
{{10, 7, 4}, {9, 7, 2, 10}, {3, 9, 8, 2, 3, 6},
 {4, 6, 6, 4, 6}, {9, 2, 5, 6, 3, 3, 1}, {1, 1, 8}, {1, 2, 5, 5},
 {3, 10, 3, 1, 5, 5, 7}, {8, 10, 1}, {6, 10, 7, 6}}
```

This is how the resulting function will look like :

```
Clear[appendSublists];
appendSublists[x_List, newelems_List] /;
   Length[x] == Length[newelems] :=
  MapThread[Append, {x, newelems}];
```

- 5.3.2.4.3 Example: deleting from each sublist of a nested list given number of elements at the beginning:

The problem to solve here is the following: given a list of lists, delete from the beginning of each sublist a number of elements given by the element of another (single) list.

To prepare the "delete" list, we first find a list of sublists lengths:

```
lengths = Length /@ result
{3, 4, 6, 5, 7, 3, 4, 7, 3, 4}
```

Then we randomly generate a number of elements to be deleted for each list, not exceeding the number of elements in it:

```
dellist = Map[Min[Random[Integer, {1, 4}], #] &, lengths]
{3, 2, 2, 2, 3, 3, 2, 1, 3, 4}
```

This is the code which does the job:

```
MapThread[Drop, {result, dellist}]
{{}, {2, 10}, {8, 2, 3, 6}, {6, 4, 6},
 {6, 3, 3, 1}, {}, {5, 5}, {10, 3, 1, 5, 5, 7}, {}, {}}
```

This is how the function will look :

```
Clear[dropFromSublists];
dropFromSublists[{sublists__List}, dellengths_List] /;
   Length[{sublists}] == Length[dellengths] :=
  MapThread[If[Length[#1] < #2, {}, Drop[#1, #2]] &,
   {{sublists}, dellengths}];
```

Check :

```
dropFromSublists[result, dellist]
{{}, {2, 10}, {8, 2, 3, 6}, {6, 4, 6},
 {6, 3, 3, 1}, {}, {5, 5}, {10, 3, 1, 5, 5, 7}, {}, {}}
```

Note the pattern used in a function : it guarantees that all the elements of the nested list are lists themselves, and the attached condition checks that the list of lengths of element sequences to be dropped has the same length as a nested list. Also, the function inside MapThread has been modified to account for cases when the instructed number of elements to be dropped is larger than the length of the sublist - in this case all elements are dropped, and an empty list is returned.

5.3.2.4.4 A digression : stricter error - checking

If this convention is unsatisfactory, and one needs a stricter condition which would issue an error message in such an event, then it is best to relegate this to patterns by modifying them appropriately :

```
Clear[dropFromSublistsStrict];
dropFromSublistsStrict[{sublists__List}, dellengths_List] :=
  MapThread[Drop, {{sublists}, dellengths}] /;
   And[Length[{sublists}] == Length[dellengths], Sequence @@
     Map[NonNegative, Map[Length, {sublists}] -dellengths]];

dropFromSublistsStrict[{sublists__List}, dellengths_List] /;
   Length[{sublists}] == Length[dellengths] :=
  "Some of the element numbers to delete larger
    than the corresponding sublist length";

dropFromSublistsStrict[{sublists__List}, dellengths_List] :=
  "The nested list and delete lengths
    list should be of the same length";
```

Note that the other possibility would be to again use the If statement inside the MapThread, which should then take the proper action when an erroneous input is encountered. But this solution is typically worse for several reasons : **a)** Some part of the evaluation would typically have happened. This may be undesirable both because time has been wasted and because side effects may have been introduced. **b)** MapThread, like many other functional constructs, can not be normally "stopped" - thus, one will have to throw an exception. **c)** A minor one: presence of If slows the function down a bit.

This sort of solution can be used when the error condition can not be determined solely by input data, but instead depends on the results already obtained in the course of execution of a given function. However, such behavior is not typical for programs designed within the functional programming paradigm (at least, in *Mathematica*), mainly because it is usually caused by side-effects such as run-time assignments or in-place data modifications.

Returning to the code above, I used the opportunity to illustrate another possible way to attach a condition to the function definition: instead of attaching it right after the "declaration" part, we can attach it at the end, after the right hand side of the definition. Also, notice once again that we have exploited the way in which *Mathematica* applies definitions (rules, associated with a function) - more specific before more general, to implement error-checking.

Be sure to understand the use of Map , Apply and Sequence in the first definition above code. Check :

```
dropFromSublistsStrict[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}, {1, 2, 3}]
{{2, 3}, {6}, {}}
dropFromSublistsStrict[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}, {1, 2, 4}]
Some of the element numbers to delete
  larger than the corresponding sublist length
dropFromSublistsStrict[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}, {1, 2, 3, 4}]
The nested list and delete
  lengths list should be of the same length
```

Of course, instead of printing error messages, any other desired action can be taken. In fact, returning error messages in the above fashion (or, worse yet, printing them with the Print command) may be ok for quick-and-dirty functions you write for yourself (say, in the exploratory stage), but not when you place your functions into packages. There exists a much better mechanism of Messages one should use in package writing. This topic is generally beyond the scope of our present discussion, so consult the books of Maeder and Wagner, as well as *Mathematica* Help and *Mathematica* Book, for details.

The additional error - checking overhead due to the pattern - matching is usually not too considerable if the patterns are mostly syntactic in nature (such as {__Integer}), or when they do some simple things like checking the list sizes. Of course, error - checking has to be done in any case if one wants to have a robust functionality. Sometimes, however, the user of the function knows for sure that the supplied data are correct (for instance, they come as a result of another function), in which case error - checking may be redundant. One possibility is to provide an option (say, with the name ArgTest) which will tell whether or not to perform the error checking :

```
fun[args__, opts___ ? OptionQ] /;
  If[ArgTest /. Flatten[{opts}] /. ArgTest → True,
   code - to - check - arguments, True] := r.h.s
```

This is however not completely safe in general, since this assumes a high level of competence (with this particular functionality) from the user. This sort of solution can be used in functions that are used only by a developer to build something on top of them.

A safer alternative would be to place several related functions in a package, so that the auxiliary ones may exchange information without redundant type checks, and the interface ones which are exported to the end user do the type checks, but only once.

■ 5.3.2.4.5 Example: rotating each sublist in a nested list differently

This is our final example of the same logic as the previous ones. Here we want to rotate the individual sublists (say, to the right) according to the list of rotations. This is our nested list

```
Clear[testlist];
testlist =
 Table[Random[Integer, {1, 10}], {10}, {Random[Integer, {2, 6}]}]
```

```
{{8, 4}, {3, 8}, {6, 8, 7, 4}, {9, 9, 2},
 {9, 8, 9, 3, 1, 3}, {7, 2, 4, 7, 4, 9}, {2, 5, 7},
 {10, 5, 9}, {4, 2, 2, 2}, {5, 10, 8, 10, 1, 6}}
```

Here is the list of rotations

```
rotatelist = Table[Random[Integer, {2, 6}], {Length[testlist]}]
```

```
{6, 4, 4, 2, 5, 2, 5, 3, 6, 3}
```

This code does the job

```
MapThread[RotateRight, {testlist, rotatelist}]
```

```
{{8, 4}, {3, 8}, {6, 8, 7, 4}, {9, 2, 9},
 {8, 9, 3, 1, 3, 9}, {4, 9, 7, 2, 4, 7}, {5, 7, 2},
 {10, 5, 9}, {2, 2, 4, 2}, {10, 1, 6, 5, 10, 8}}
```

The function will look like

```
Clear[rotateSublists];
rotateSublists[{sublists__List}, {rotatenums__Integer}] /;
   Length[{sublists}] == Length[{rotatenums}] :=
  MapThread[RotateRight, {{sublists}, {rotatenums}}];
```

As we will see in chapter VI, for the problem of fast extraction of all matrix diagonals, the most efficient solution will be based on this example.

■

All of the three examples above can be also done using Map and Apply, and pure functions, along the lines outlined above. We leave it as an exercise to the reader to implement these versions. The point however is that, as we also already discussed, doing it with Thread or MapThread may be several times more efficient. In fact, this should be precisely the reason why these operations were implemented as separate commands - in practice they are needed quite frequently, and using Map in such situations in the symbolic environment of *Mathematica* may induce significant overhead.

■ 5.3.2.4.6 Example: imitating Transpose

Consider the following example:

```
MapThread[List, {Range[10], Range[11, 20]}]
```

```
{{1, 11}, {2, 12}, {3, 13}, {4, 14}, {5, 15},
 {6, 16}, {7, 17}, {8, 18}, {9, 19}, {10, 20}}
```

If we look carefully at the result, we realize that what we obtained is a list Transposed with respect to the original input. Indeed:

```
Transpose[{Range[10], Range[11, 20]}]
```

```
{{1, 11}, {2, 12}, {3, 13}, {4, 14}, {5, 15},
 {6, 16}, {7, 17}, {8, 18}, {9, 19}, {10, 20}}
```

Thus, we may imitate the action of the Transpose command:

```
Clear[myTranspose];
myTranspose[x_List] := MapThread[List, x];
```

We can now compare the performance

```
perftest = Table[i + j, {i, 20}, {j, 30}];
```

```
myTranspose[perftest] // myTiming
```

```
0.00016
```

```
Transpose[perftest] // myTiming
```

```
0.0000201
```

We come to the usual conclusion that the built-in operations are favored. However, there is more to it in this particular example. What we compare here is not just two realizations - ours vs built-in, but two styles of programming: functional vs. structure operations. The typical commands of the latter style are: Transpose, Partition, Flatten, perhaps also Inner , Outer (although we cover them in this chapter) , RotateRight, RotateLeft, and some others. All these commands are extremely fast and efficient. So, if any of these can be used, they are usually favored with respect to a functional realization, which, in terms of efficiency, comes next. The rule-based realization usually comes after functional, and the procedural realization is very often the last in our performance winners list.

Heuristically, this can be understood as follows. Structural operations are the most precise, in the sense that the structures they operate on are rather specific (for instance, Transpose requires sublists of the same length, etc). Also, their role is mostly in rearranging the existing structure, but not transforming the pieces. Functional style is still quite precise (since one needs to have a clear idea of the structure of an expression before using Map and Apply), but somewhat less restrictive. Also, here we can transform pieces by Mapping and Applying functions to them. The rule-based approach is less precise in the sense that we don't need to know beforehand where in the expression the rule applies - if we construct the rule correctly, it will apply to all places where needed. The overhead induced here is mostly due to the pattern-matcher which has to determine for us the correct places where to apply the rules. I hastily comment that there are cases when rule-based approach is extremely efficient, but this usually means that rules and patterns are very well matched to expressions they operate on, by a programmer who has a very good and precise idea of how these rules will be applied (so that the pattern-matcher "wastes" as little time as possible on false matches). Finally, in procedural approach we don't use the natural advantage of many *Mathematica'*s functions which work with whole expressions, but break them into pieces (e.g. array indexing), which means that we are going in directions entirely orthogonal to those for which the system has been optimized.

```
Clear[myTranspose, perftest];
```

5.3.3 Inner

As it is nicely stated in *Mathematica* Help, Inner is a generalization of an inner product. The format of the command is

```
Inner[f, list1, list2, g]
```

The lists <list1> and <list2> have to be of the same length. The function f plays a role of multiplication, and g - of addition.

5.3.3.1 Simple examples

```
Inner[f, {a, b, c}, {d, e, f}, g]
g[f[a, d], f[b, e], f[c, f]]
```

We can get back a standard inner product for a vector:

```
Inner[Times, {a, b, c}, {d, e, f}, Plus]
a d + b e + c f
```

5.3.3.2 Example: Imitating Inner

We can imitate the workings of Inner with MapThread and Apply in the following manner:

Inner[f,list1,list2,g]==Apply[g,MapThread[f,{list1,list2}]]

where the equality sign means "acts similarly". For instance:

```
g @@ MapThread[f, {{a, b, c}, {d, e, f}}]
g[f[a, d], f[b, e], f[c, f]]
```

Alternatively,

Inner[f,list1,list2,g]==Apply[g,f@@@Transpose[{list1,list2}]]

```
Apply[g, f @@@ Transpose[{{a, b, c}, {d, e, f}}]]
g[f[a, d], f[b, e], f[c, f]]
```

It is good to realize that Inner is in some sense a more specialized function, than say Thread or MapThread (it takes only two lists, for instance, and they have to have the same length). This means that in certain situations, we can expect it to give a better performance.

5.3.3.3 Example: Creating a list of rules

Here, for example, we can use Inner to create a list of rules:

```
Inner[Rule, Range[10], Range[11, 20], List]
{1 → 11, 2 → 12, 3 → 13, 4 → 14,
 5 → 15, 6 → 16, 7 → 17, 8 → 18, 9 → 19, 10 → 20}
```

The function will look like

```
Clear[createRules];
createRules[lhs_List, rhs_List] /; Length[lhs] == Length[rhs] :=
 Inner[Rule, lhs, rhs, List]
```

We can compare its performance with that of Thread:

```
Inner[Rule, Range[100], Range[101, 200], List] // myTiming
0.00023
MapThread[Rule, {Range[100], Range[101, 200]}] // myTiming
0.000341
```

By using Inner in this case, we get about 30% increase in performance.

- 5.3.3.4 Example: Comparing two lists

As another example, here we use Inner to compare two lists and return positions where the elements of the two lists are different:

```
Inner[Equal, {1, 2, 3, 4, 5, 6},
 {1, 2, 1, 4, 2, 6}, Position[{##}, False] &]


{{3}, {5}}
```

we can express this as a function:

```
Clear[compareLists];
compareLists[list1_List, list2_List] /;
  Length[list1] == Length[list2] :=
 Inner[SameQ, list1, list2, Position[{##}, False] &]
```

check:

```
compareLists[{1, 2, 3, 4, 5, 6}, {1, 2, 1, 4, 2, 6}]
{{3}, {5}}
```

As an alternative, we may consider such an implementation:

```
compareListsAlt[list1_List, list2_List] /;
  Length[list1] == Length[list2] :=
 Position[Abs[list1 - list2], _ ? Positive]
```

check:

```
compareListsAlt[{1, 2, 3, 4, 5, 6}, {1, 2, 1, 4, 2, 6}]
{{3}, {5}}
```

The last one is based on exploiting the fast Subtract operation which operates on entire lists. We can compare the performance:

```
compareLists[Range[1000], Range[1000]] // myTiming
0.0015
compareListsAlt[Range[1000], Range[1000]] // myTiming
0.00231
```

The result is interesting. In the Inner-based implementation, the most expensive operation is to thread Equal on a list of pairs of elements of our original lists (that's what it does internally). In the implementation based on subtraction, the most expensive is Position with the <_?Positive> pattern, and it turns out to be slower. In addition to this, the Subtract - based implementation is less general since it will work correctly only on numeric lists - otherwise the <_?Positive> pattern will not match.

- 5.3.3.5 Example: reconstructing a number from its factorized form

Say, we are given some number, like for instance:

```
num = 3 628 800
3 628 800
```

Let us factorize it, using the built-in command FactorInteger:

```
factored = FactorInteger[num]
{{2, 8}, {3, 4}, {5, 2}, {7, 1}}
```

In each sublist, the first number is a base, and the second - an exponent (the power). Now, we want to perform the opposite operation: reconstruct the number back from its factorized form. It is clear that the idiom of Inner matches this problem in principle. What we have to do though is to Transpose the initial list, and then Apply Sequence to it:

```
Inner[Power,Sequence@@Transpose[factored],Times]
3 628 800
```

We can now write a function:

```
Clear[multiplyFactored]
multiplyFactored[fact_List] :=
  Inner[Power, Sequence @@ Transpose[fact], Times];
```

I leave it as an exercise to the reader to add the condition to check that the input list contains sublists of the same length. We now check:

```
multiplyFactored[FactorInteger[100]]
100
```

For this problem, there exists an alternative solution in terms of core functions Map and Apply:

```
Clear[multiplyFactoredAlt];
multiplyFactoredAlt[fact_List] := Apply[Times, Power @@@ fact];
```

Make sure you understand the code. This solution is more concise, and we may expect that it has somewhat better performance. We can check it:

First check that it works:

```
multiplyFactoredAlt[factored]
3 628 800
```

This will be our test factorized number (50!)

```
testfact = FactorInteger[50 !]
{{2, 47}, {3, 22}, {5, 12}, {7, 8}, {11, 4}, {13, 3}, {17, 2}, {19, 2},
 {23, 2}, {29, 1}, {31, 1}, {37, 1}, {41, 1}, {43, 1}, {47, 1}}
```

We now test:

```
multiplyFactored[testfact] // myTiming
0.0000731
multiplyFactoredAlt[testfact] // myTiming
0.0000501
```

This shows that sometimes the core functions give a more direct solution, which make us once again appreciate their usefulness and versatility.

```
Clear[num, factored, testfact,
  multiplyFactored, multiplyFactoredAlt];
```

■ 5.3.4 Outer

This is another very useful and widely used function. It takes several lists and basically creates all possible combinations of the elements of different input lists (Cartesian product). Then it can apply some function to these combinations. The format of the command in the simplest form is:

```
Outer[function, list1, list2, ...]
```

■ 5.3.4.1 Simple examples

```
Outer[List, {a, b}, {c, d}]
{{{a, c}, {a, d}}, {{b, c}, {b, d}}}
Outer[List, {a, b}, {c, d}, {e, f}]
{{{{a, c, e}, {a, c, f}}, {{a, d, e}, {a, d, f}}},
 {{{b, c, e}, {b, c, f}}, {{b, d, e}, {b, d, f}}}}
```

As you can see, the result is a nested list where innermost sublists correspond to sweeping through the rightmost of the input lists, and so on.The lists are not necessarily of the same length:

```
Outer[f, {a, b}, {c, d, e}]
{{f[a, c], f[a, d], f[a, e]}, {f[b, c], f[b, d], f[b, e]}}
```

We can use Outer for construction of certain matrices

```
Outer[f, {1, 2, 3}, {4, 5}] // MatrixForm
```

$$\begin{pmatrix} f[1,4] & f[1,5] \\ f[2,4] & f[2,5] \\ f[3,4] & f[3,5] \end{pmatrix}$$

■ 5.3.4.2 Example: natural numbers

This creates first 100 natural numbers (if we count 0 as one)

```
Flatten[Outer[#1 * 10 + #2 &, Range[0, 9], Range[0, 9]]]
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51,
 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
```

■ 5.3.4.3 Example: binary numbers

This gives binary forms of numbers 0-31. Note the Flatten operator - it is a frequent companion of Outer.

```
Flatten[Outer[List, Sequence @@ Table[{0, 1}, {5}]], 4]
```

```
{{0, 0, 0, 0, 0}, {0, 0, 0, 0, 1}, {0, 0, 0, 1, 0}, {0, 0, 0, 1, 1},
 {0, 0, 1, 0, 0}, {0, 0, 1, 0, 1}, {0, 0, 1, 1, 0}, {0, 0, 1, 1, 1},
 {0, 1, 0, 0, 0}, {0, 1, 0, 0, 1}, {0, 1, 0, 1, 0}, {0, 1, 0, 1, 1},
 {0, 1, 1, 0, 0}, {0, 1, 1, 0, 1}, {0, 1, 1, 1, 0}, {0, 1, 1, 1, 1},
 {1, 0, 0, 0, 0}, {1, 0, 0, 0, 1}, {1, 0, 0, 1, 0}, {1, 0, 0, 1, 1},
 {1, 0, 1, 0, 0}, {1, 0, 1, 0, 1}, {1, 0, 1, 1, 0}, {1, 0, 1, 1, 1},
 {1, 1, 0, 0, 0}, {1, 1, 0, 0, 1}, {1, 1, 0, 1, 0}, {1, 1, 0, 1, 1},
 {1, 1, 1, 0, 0}, {1, 1, 1, 0, 1}, {1, 1, 1, 1, 0}, {1, 1, 1, 1, 1}}
```

■ 5.3.4.4 Example: table of values for trigonometric functions

Here we will use Outer to create a table of values of the main 4 trigonometric functions for various typical values of the argument. Here are our functions:

```
functions = {Sin, Cos, Tan, Cot}
```

```
{Sin, Cos, Tan, Cot}
```

And the values of the angle:

```
args = {0, Pi / 6, Pi / 4, Pi / 2}
```

$$\left\{0, \frac{\pi}{6}, \frac{\pi}{4}, \frac{\pi}{2}\right\}$$

Here is a table of values:

```
values = Outer[#2[#1] &, args, functions]
```

$$\left\{\{0, 1, 0, \text{ComplexInfinity}\}, \left\{\frac{1}{2}, \frac{\sqrt{3}}{2}, \frac{1}{\sqrt{3}}, \sqrt{3}\right\}, \left\{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 1, 1\right\}, \{1, 0, \text{ComplexInfinity}, 0\}\right\}$$

Now we will add names of functions and values of the argument, for displaying purposes.

```
info = Transpose[Prepend[
   Transpose[Prepend[values, functions]], Join[{"\\"}, args]]]
```

$$\left\{\{\backslash, \text{Sin}, \text{Cos}, \text{Tan}, \text{Cot}\}, \{0, 0, 1, 0, \text{ComplexInfinity}\}, \left\{\frac{\pi}{6}, \frac{1}{2}, \frac{\sqrt{3}}{2}, \frac{1}{\sqrt{3}}, \sqrt{3}\right\}, \left\{\frac{\pi}{4}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 1, 1\right\}, \left\{\frac{\pi}{2}, 1, 0, \text{ComplexInfinity}, 0\right\}\right\}$$

Finally, we display the table

```
TableForm[info] /. ComplexInfinity → ∞ // TraditionalForm
```

$$\begin{array}{lllll}
\backslash & \text{Sin} & \text{Cos} & \text{Tan} & \text{Cot} \\
0 & 0 & 1 & 0 & \infty \\
\frac{\pi}{6} & \frac{1}{2} & \frac{\sqrt{3}}{2} & \frac{1}{\sqrt{3}} & \sqrt{3} \\
\frac{\pi}{4} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 & 1 \\
\frac{\pi}{2} & 1 & 0 & \infty & 0
\end{array}$$

- 5.3.4.5 Example: creating interpolations for functions of several variables

Say, we have a function of two variables, for instance

```
Clear[f];
f[x_, y_] := Sin[2 Sqrt[x^4 + y^4]]
```

We want to get an interpolation of this function on a rectangular grid $0 \leq x \leq 2, 0 \leq y \leq 2$, with a step 0.4 in x direction and 0.5 in y direction. We first create one-dimensional grids:

```
xgrid = Range[0., 2., 0.4]
```

```
{0., 0.4, 0.8, 1.2, 1.6, 2.}
```

```
ygrid = Range[0., 2., 0.5]
```

```
{0., 0.5, 1., 1.5, 2.}
```

Now we use Outer to construct the values of the function on all possible combinations of the points in these two grids:

```
(vals = Outer[{#1, #2, f[#1, #2]} &, xgrid, ygrid]) // Short[#, 7] &
```

```
{{{0., 0., 0.}, {0., 0.5, 0.479426}, {0., 1., 0.909297},
  {0., 1.5, -0.97753}, {0., 2., 0.989358}}, <<4>>, {<<1>>}}
```

We used here the pure function {#1,#2,f[#1,#2]}&, since we also need the coordinates of the point on the 2D grid, in addition to the value of the function. We now have to use Flatten, to remove one layer of internal curly (list) braces:

```
Flatten[vals, 1] // Short[#, 5] &
```

```
{{0., 0., 0.}, {0., 0.5, 0.479426},
 <<26>>, {2., 1.5, 0.243524}, {2., 2., -0.949821}}
```

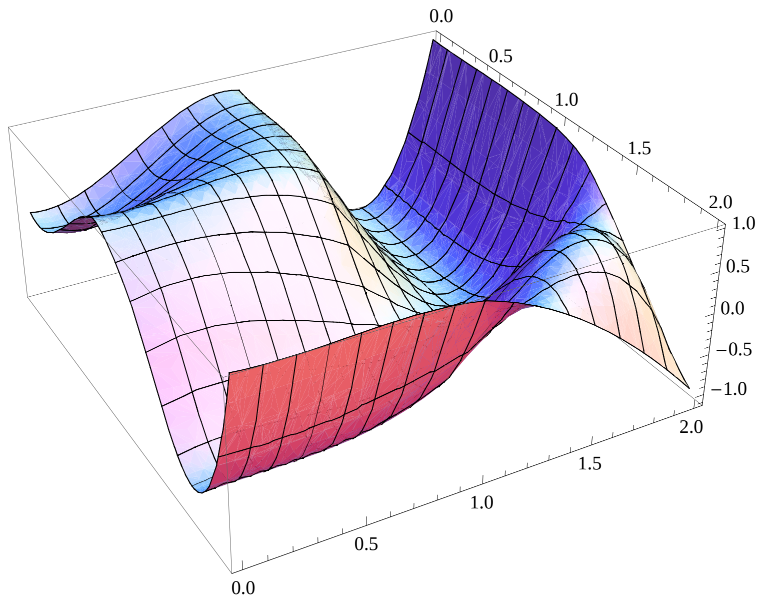
Now we can use the Interpolation command on these values:

```
intfun = Interpolation[Flatten[vals, 1]]
```

```
InterpolatingFunction[{{0., 2.}, {0., 2.}}, <>]
```

We can use Plot3D to visualize our function:

```
Plot3D[intfun[x, y], {x, 0, 2}, {y, 0, 2}]
```



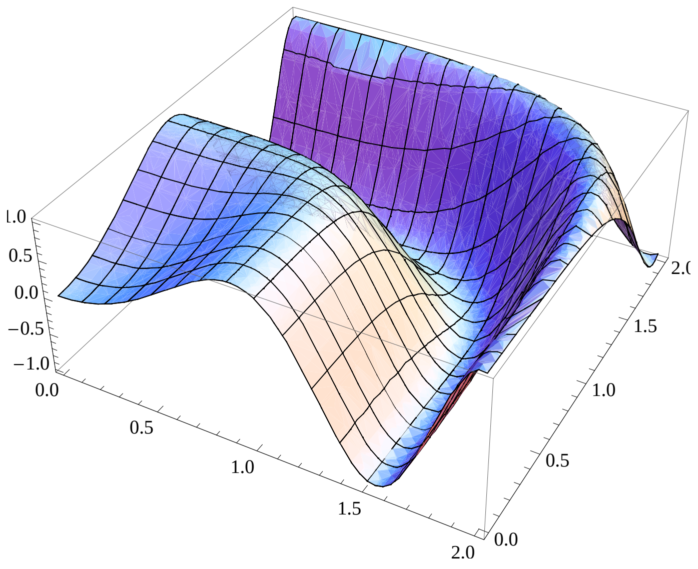
We obviously considered a grid too coarse to grasp all important details in the behavior of our function. Let us now create a function which will take a name of the function to be interpolated, the list of {start,end,step} for each direction, and return an interpolated function:

```
Clear[giveInterpolated];
giveInterpolated[fn_Symbol, xpars_List, ypars_List] :=
  Interpolation[Flatten[Outer[{#1, #2, fn[#1, #2]} &,
     Apply[Sequence, Range @@@ {xpars, ypars}]], 1]];
```

Here we need to Apply Sequence since Outer receives a sequence of lists. Range@@@{xpars,ypars} creates a list of two grids.

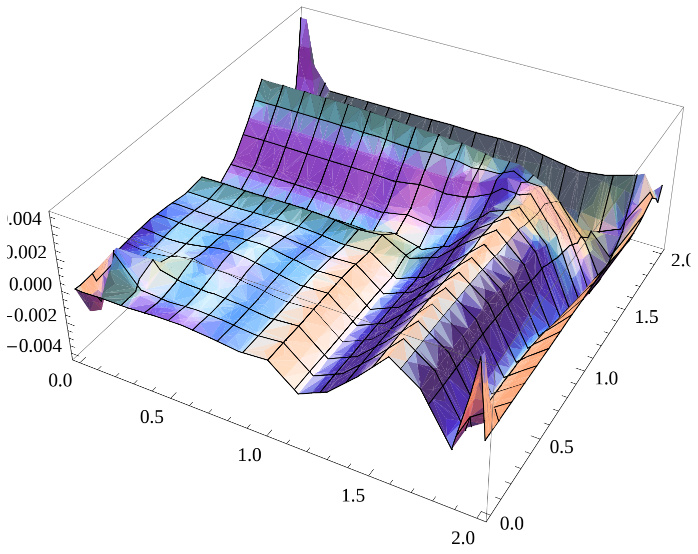
And now let us use more points:

```
newintfun = giveInterpolated[f, {0., 2., 0.1}, {0., 2., 0.1}]
InterpolatingFunction[{{0., 2.}, {0., 2.}}, <>]
Plot3D[newintfun[x, y], {x, 0, 2}, {y, 0, 2}]
```



We can see how well our interpolation approximates the original function

```
Plot3D[newintfun[x, y] -f[x, y], {x, 0, 2}, {y, 0, 2}]
```



Notice that here we created our own higher-order function, since it takes a name of another function as one of its arguments.

The same method will work for functions with more variables. The only real change will be that we will have to Flatten the list of values to more depth.

```
Clear[f, xgrid, ygrid, intfun, newintfun, giveInterpolated];
```

■ 5.3.4.6 Example: imitating Outer

In this example, we will try to imitate Outer in the case when we have only two lists, with the functions that we already know. Let us start with a sub-problem: given a list of elements and an object, form all the pairs of this object with the elements of the list. For example, our list will be {a,b,c}, and our stand-alone element will be d. Then the solution will be:

```
Clear[a, b, c, d, e, f];
Thread[List[{a, b, c}, d]]
```

```
{{a, d}, {b, d}, {c, d}}
```

To create all possible combinations of elements of the two lists, say {a,b,c} and {d,e,f}, we have to repeat this operation with the same list {a,b,c} but different elements (d,e, and f). This means that we have to do something like this:

```
Map[Thread[List[{a, b, c}, #]] &, {d, e, f}]
```

```
{{{a, d}, {b, d}, {c, d}},
 {{a, e}, {b, e}, {c, e}}, {{a, f}, {b, f}, {c, f}}}
```

Finally, we have to generalize to any head, which amounts to substituting List in the code by that head (this is why I used here the literal head List instead of {} from the beginning - to make this transition natural):

```
Map[Thread[g[{a, b, c}, #]] &, {d, e, f}]
```

```
{{g[a, d], g[b, d], g[c, d]},
 {g[a, e], g[b, e], g[c, e]}, {g[a, f], g[b, f], g[c, f]}}
```

Thus, our final function will look like:

```
Clear[myOuter];
myOuter[g_, list1_List, list2_List] :=
  Map[Thread[g[#, list2]] &, list1];
```

I have interchanged list1 and list2, and also the order of arguments inside g, to get exactly the same output that Outer gives, where the innermost sublists in the output list correspond to the sweeping through the rightmost input list.

Check:

```
myOuter[g, {a, b, c}, {d, e, f}]
```

```
{{g[a, d], g[a, e], g[a, f]},
 {g[b, d], g[b, e], g[b, f]}, {g[c, d], g[c, e], g[c, f]}}
```

```
Outer[g, {a, b, c}, {d, e, f}]
```

```
{{g[a, d], g[a, e], g[a, f]},
 {g[b, d], g[b, e], g[b, f]}, {g[c, d], g[c, e], g[c, f]}}
```

Let us check the performance:

```
myOuter[List, Range[15], Range[20]] // myTiming
0.000271
Outer[List, Range[15], Range[20]] // myTiming
0.00018
```

We are actually fairly close (at least for these lists), given that Outer is a built-in function.

For completeness, let me mention that there exists (at least one more) solution with comparable (slightly worse) performance. This solution may be obtained along the lines discussed above, where we noticed that whenever a function is Mapped onto a list with some arguments fixed, this can be rewritten using MapThread. Here is the solution:

```
Clear[myOuter1];
myOuter1[g_, list1_List, list2_List] := MapThread[
   Thread[g[##]] &, {list1, Table[list2, {Length[list1]}]}];
```

I leave it as an exercise to the reader to figure out precisely how it works. For now, let us check:

```
myOuter1[g, {a, b}, {d, e, f}]
{{g[a, d], g[a, e], g[a, f]}, {g[b, d], g[b, e], g[b, f]}}
myOuter1[List, Range[15], Range[20]] // myTiming
0.000311
Clear[myOuter, myOuter1];
```

- 5.3.4.7 Case study: creating ordered subsets for a given set

Creating ordered pairs

We can use Outer to create all possible pairs of elements in a given list. For example:

```
pairs = Flatten[Outer[List, {a, b, c}, {a, b, c}], 1]
{{a, a}, {a, b}, {a, c}, {b, a}, {b, b}, {b, c}, {c, a}, {c, b}, {c, c}}
```

This is almost the same as all ordered subsets of length 2 - the difference is that we have also lists of identical elements. The latter may easily be eliminated in this case:

```
DeleteCases[pairs, {z_, z_}]


{{a, b}, {a, c}, {b, a}, {b, c}, {c, a}, {c, b}}
```

Let us now write a general function which will create all ordered subsets for a given set (list) of distinct elements. As a first step, we will re-package our code into a function (for ordered pairs)

```
Clear[orderedPairs];
orderedPairs[set_List] :=
 DeleteCases[Flatten[Outer[List, set, set], 1], {z_, z_}]
```

check:

```
orderedPairs[{a}]
{}
orderedPairs[{a, b}]
{{a, b}, {b, a}}
orderedPairs[{a, b, c, d}]
{{a, b}, {a, c}, {a, d}, {b, a}, {b, c},
 {b, d}, {c, a}, {c, b}, {c, d}, {d, a}, {d, b}, {d, c}}
```

Generalizing to ordered subsets

To generalize to higher tuples, we need to duplicate <set> in Outer several times, to Flatten to a deeper level, and to use a more complicated pattern:

```
Clear[orderedSubsets];
orderedSubsets[set_List, order_Integer] :=
 DeleteCases[Flatten[Outer[List, Sequence @@ Table[set, {order}]],
   order - 1], {x___, t_, y___, t_, z___}]
```

Check now:

```
orderedSubsets[{a, b, c, d}, 2]
{{a, b}, {a, c}, {a, d}, {b, a}, {b, c},
 {b, d}, {c, a}, {c, b}, {c, d}, {d, a}, {d, b}, {d, c}}
orderedSubsets[{a, b, c, d}, 3]
{{a, b, c}, {a, b, d}, {a, c, b}, {a, c, d}, {a, d, b}, {a, d, c},
 {b, a, c}, {b, a, d}, {b, c, a}, {b, c, d}, {b, d, a}, {b, d, c},
 {c, a, b}, {c, a, d}, {c, b, a}, {c, b, d}, {c, d, a}, {c, d, b},
 {d, a, b}, {d, a, c}, {d, b, a}, {d, b, c}, {d, c, a}, {d, c, b}}
orderedSubsets[{a, b, c, d}, 4]
{{a, b, c, d}, {a, b, d, c}, {a, c, b, d}, {a, c, d, b},
 {a, d, b, c}, {a, d, c, b}, {b, a, c, d}, {b, a, d, c}, {b, c, a, d},
 {b, c, d, a}, {b, d, a, c}, {b, d, c, a}, {c, a, b, d}, {c, a, d, b},
 {c, b, a, d}, {c, b, d, a}, {c, d, a, b}, {c, d, b, a}, {d, a, b, c},
 {d, a, c, b}, {d, b, a, c}, {d, b, c, a}, {d, c, a, b}, {d, c, b, a}}
orderedSubsets[{a, b, c, d}, 5]
{}
```

In the last case, the result is an empty list due to the pigeonhole principle: we have only 4 distinct elements and are trying to create subsets of length 5, which means that in any such subset at least two elements will be the same, and it is then eliminated by DeleteCases.

Efficiency analysis

Even though our code works correctly, it does not work efficiently. For instance:

```
orderedSubsets[{a, b, c, d}, 7] // Timing
{0.31 Second, {}}
```

This is because, a huge list is created first to be completely eliminated later. We will be better off by adding an explicit condition:

```
Clear[orderedSubsets];

orderedSubsets[set_List, order_Integer] /; order ≤ Length[set] :=
  DeleteCases[Flatten[Outer[List, Sequence @@ Table[set, {order}]],
    order - 1], {x___, t_, y___, t_, z___}];

orderedSubsets[set_List, order_Integer] /; order > Length[set] = {};
```

As usual, we may ask is how efficient is our implementation.The main source of inefficiency here is that many of the combinations generated will have identical elements and will then be deleted later. It would be better if they were not generated from the beginning. Thus, in terms of this factor, our implementation is rather efficient for ordered pairs and large <set>, but completely inefficient for subsets of length comparable to the length of initial set itself.

Improving orderedPairs

Another suspected source of inefficiency is the pattern-matching in DeleteCases. For ordered pairs, we can eliminate the pattern-matching stage the help of MapThread:

```
Clear[orderedPairsNew];
orderedPairsNew[set_List] := Flatten[MapThread[Drop,
    {Outer[List, set, set], Map[List, Range[Length[set]]]}], 1];
```

What happens is that first, second, etc elements are dropped from first, second, etc sublists of a list generated by Outer. These are exactly the elements containing duplicates. Make sure you understand how the code works. Check:

```
orderedPairsNew[{a, b, c, d}]
{{a, b}, {a, c}, {a, d}, {b, a}, {b, c},
 {b, d}, {c, a}, {c, b}, {c, d}, {d, a}, {d, b}, {d, c}}
```

We can now check how much did we gain if at all:

```
orderedPairs[Range[70]] // myTiming
0.013
```

```
orderedPairsNew[Range[70]] // myTiming
0.0032
```

We see that we get a 2-3 times difference which is substantial (this factor is not constant. It will be less for smaller sets and larger for larger sets). Thus, this is currently our best implementation of the ordered pairs.

For the case of general subsets, there is no point in checking, since we already did the analysis and found that our implementation is inefficient. Can we find a better one?

A better overall implementation

Let us try to find an alternative implementation for the ordered subsets function. One possibility is the following: there is a built-in function Subsets, which generates all distinct subsets of a given size. All that remains is to create all permutations for any of the subsets generated. Another built-in command Permutations will help us with this. So, let us start with the test set, for instance

```
Clear[a, b, c, d];
testset = {a, b, c, d}
```

```
{a, b, c, d}
```

Now, let us find say all subsets of length 3:

```
Subsets[testset, {3}]
```

```
{{a, b, c}, {a, b, d}, {a, c, d}, {b, c, d}}
```

Let us pick one of them, say a first one. To make all the permutations, we use the Permutations command:

```
Permutations[{a, b, c}]
```

```
{{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}}
```

All that remains to be done is to Map Permutations on the list generated by Subsets, and then Flatten the latter

```
Flatten[Permutations /@ Subsets[testset, {3}], 1]
```

```
{{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a},
 {a, b, d}, {a, d, b}, {b, a, d}, {b, d, a}, {d, a, b}, {d, b, a},
 {a, c, d}, {a, d, c}, {c, a, d}, {c, d, a}, {d, a, c}, {d, c, a},
 {b, c, d}, {b, d, c}, {c, b, d}, {c, d, b}, {d, b, c}, {d, c, b}}
```

We expect this implementation to be vastly superior to the previous one, due to a more direct use of built-in commands, but most of all, the fact that we avoided creation of large number of elements which then have to be deleted. Let us package this solution into a function:

```
Clear[orderedSubsetsNew];
orderedSubsetsNew[set_List, order_Integer] :=
  Flatten[Map[Permutations, Subsets[set, {order}], 1]];
```

Let me remark that there is also some notion of beauty or aesthetics which we can assign to the implementations in *Mathematica*. This implementation is certainly more beautiful than the previous one (at least, in my taste).

Let us compare the performance in the case where our solution is not that bad - for ordered pairs:

```
orderedSubsetsNew[Range[70], 2] // myTiming
0.00491
orderedPairsNew[Range[70]] // myTiming
0.0039
```

We observe that for ordered pairs, our specialized solution based on Outer is slightly better than an implementation based on Subsets-Permutations pair. However, already for 3-tuples our general <orderedSubsets> function is hopelessly slower:

```
orderedSubsets[Range[70], 3] // myTiming
2.944
orderedSubsetsNew[Range[70], 3] // myTiming
0.241
```

This looks like quite a long execution time even for the better solution. But let us see how many combinations (3-tuples) have been produced:

```
orderedSubsetsNew[Range[70], 3] // Length
985 320
```

We see that for the general case (not just ordered pairs), the Outer-based solution is miles away from the Subsets-Permutations based one. The main reason is of course that while the Outer was a possible choice, it was not exactly the right idiom in this case. It produces a lot of combinations that have to be eliminated later, which means that this is just a bad algorithm for general tuples (but reasonable for pairs).

```
Clear[orderedPairs, orderedPairsNew, orderedSubsets, pairs, testset];
```

■ 5.3.4.8 Using Outer in more complicated cases: a caution

Outer may be used in more general setting, in particular when the input lists are not simple, but nested lists. There is one specific instance of that case which I would like to discuss now.

```
Clear[f];
```

Consider the following situation:

```
Outer[f, {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}]
{{{{f[1, 5], f[1, 6]}, {f[1, 7], f[1, 8]}},
  {{f[2, 5], f[2, 6]}, {f[2, 7], f[2, 8]}}},
 {{{f[3, 5], f[3, 6]}, {f[3, 7], f[3, 8]}},
  {{f[4, 5], f[4, 6]}, {f[4, 7], f[4, 8]}}}}
```

This output is not what one would immediately expect. What if I want to get my function <f> applied to the sublists, like {{f[{1,2},{5,6}],...}}. To achieve this, we have to tell Outer that it should treat sublists as individual elements. This is done by specifying elements on which level of the input lists (first in this case) should be treated as individual elements:

```
Outer[f, {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}, 1]
```

```
{{f[{1, 2}, {5, 6}], f[{1, 2}, {7, 8}]},
 {f[{3, 4}, {5, 6}], f[{3, 4}, {7, 8}]}}
```

Now we get what we wanted. In this particular case, another possibility to get it is to use **Distribute**:

```
Distribute[f[{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}], List]
```

```
{f[{1, 2}, {5, 6}], f[{1, 2}, {7, 8}],
 f[{3, 4}, {5, 6}], f[{3, 4}, {7, 8}]}
```

For more details on Distribute, consult *Mathematica* Help and *Mathematica* Book.

5.4 Nest Family

5.4.1 Nest and NestList

This function is used to repeatedly apply the same function on an expression. The format is :

```
Nest[function, expression, n],
```

where <n > should be an integer giving the number of times that the function has to be applied. For example:

5.4.1.1 Simple examples

```
ClearAll[f, x];
Nest[f, x, 5]
```

```
f[f[f[f[f[x]]]]]
```

Consider, for instance,

```
f[x_] := x^2;
```

```
Nest[f, 2, 3]
```

```
256
```

5.4.1.2 NestList

The function NestList is really the same as Nest but it gives more information, since its output is a list of all intermediate steps of the application of Nest. For the above example:

```
NestList[f, 2, 3]
```

```
{2, 4, 16, 256}
```

```
NestList[f, x, 3]
```

$\{x, x^2, x^4, x^8\}$

We also see that the first element in the NestList is always the original expression, which corresponds to the function <f> applied zero times.

It is important that NestList is as efficient as Nest - there is no penalty for getting all the intermediate results. Indeed, the function still has to apply in stages - once, twice, etc - so the intermediate results are in principle internally available to the system. Simple Nest just does not collect them.

5.4.1.3 Pure functions

Both Nest and NestList work with pure functions as well:

```
NestList[#^2 &, 2, 3]
```

```
{2, 4, 16, 256}
```

■ 5.4.1.4 Example: imitating Nest

It is not at all difficult to write our own version of Nest. And this is perhaps one of the rare cases where the procedural programming style is quite good:

```
myNest[f_, x_, n_Integer] :=
  Module[{var = x}, Do[var = f[var], {n}]; var];
```

Let us check:

```
Clear[f];
myNest[f, x, 5]
```

```
f[f[f[f[f[x]]]]]
```

Our function will also work with pure functions:

```
myNest[#^2 &, x, 4]
```

x^{16}

Let us compare the performance with that of a built-in one:

```
myNest[#^1.001 &, 2, 100]
```

```
2.15116
```

```
Nest[#^1.001 &, 2, 100]
```

```
2.15116
```

```
myNest[#^1.001 &, 2, 100] // myTiming
```

```
0.000511
```

```
Nest[#^1.001 &, 2, 100] // myTiming
```

```
0.0000701
```

We still have several times difference in this example.

The reason that I insert the rather boring performance comparisons in so many places is to point out one single thing: try to avoid writing your own functions if you can find a better idiom to solve your problem, which matches some of the built-in ones. The fact that we can rewrite most of the built-in functions and imitate their behavior shows once again that in some sense *Mathematica*'s language is overcomplete. Why, then, all these extra functions were written? The answer is simple: to give better performance in certain cases. Also, note that while the functions like Nest and others considered in this chapter are in some sense specific, they are on the other hand quite abstract and then can handle a lot of different problems. The trick is to learn to translate your given problem into a right *Mathematica* idiom.

One can argue that in other languages like C one can always start from scratch, write any such "derivative" function with very few initial building blocks, and be sure that it will give a reasonable performance. But this is just not so given a real level of abstraction that functions like Nest, Thread, Outer, etc can handle - they can work on essentially any objects without any modification. And this leads to another important consequence: the well-written *Mathematica* code is usually very concise, more so than in most other programming languages. But as Paul Graham has put it, "succinctness is power" [14].

5.4.1.5 Example: approximating the square root of a number

Nest is well-suited to be used with recursive functions (in the mathematical sense). For example, for the approximate computation of the square root of some number A, one may use a sequence :

$t_{n+1} = 1/2\,(t_n + A/t_n)$

We can define a function which will do this transformation. Let us start with some fixed number, say 3. Then we can use a pure function:

```
NestList[(# + 3 / #) / 2 &, 5., 5]
{5., 2.8, 1.93571, 1.74276, 1.73208, 1.73205}
```

Here our starting number was 5, and we used 5 iterations altogether. The list of intermediate results shows that this method converges quite fast. If we are interested in final result only, then we use Nest:

```
result = Nest[(# + 3 / #) / 2 &, 5., 5]
1.73205
```

We can see how close we are:

```
result^2 - 3
1.08461 × 10^-9
```

Now, we would like to be able to indicate the number <A> from the beginning. One way is to make a function like this:

```
Clear[mySquareRoot];
mySquareRoot[number_ ? NumericQ, iternum_Integer ? Positive] :=
  Nest[(# + number / #) / 2 &, 1., iternum];
```

Here we adopt a convention that our approximate solution always starts from 1. By using a more elaborate starting point which will depend on the number A, one may reduce somewhat the number of iterations needed, but the convergence is quite fast anyway.

There are two interesting details in the above code. The first is that the parameter passed to the function through the pattern-defined definition gets then embedded into a pure function inside Nest. This possibility is very often useful.

The second is the use of <NumericQ> predicate. It gives true on any object on which the application of *Mathematica* <N> command results in a number. For instance,

```
NumericQ[Pi]
True
```

There is another predicate of the similar type - <NumberQ>. This one however is restricted to numbers only:

```
NumberQ /@ {Pi, 2}
{False, True}
```

Let us now check our function, by Mapping it on a list of numbers:

```
reslist = mySquareRoot[#, 5] & /@ Range[10]
```

```
{1., 1.41421, 1.73205, 2., 2.23607,
 2.44949, 2.64575, 2.82843, 3., 3.16228}
```

```
reslist^2
```

```
{1., 2., 3., 4., 5., 6., 7., 8., 9., 10.}
```

There is another solution to the problem above - to write a function that will automatically embed the number <A> into a pure function, and then use this function in Nest. But then, we have to write a function that returns a pure function. Is this possible? Well, why not:

```
Clear[iterFun];
iterFun[number_ ? NumericQ] := (# + number / #) / 2 &;
```

We now rewrite our square root function:

```
Clear[mySquareRootNew];
mySquareRootNew[number_ ? NumericQ, iternum_Integer ? Positive] :=
  Nest[iterFun[number], 1., iternum];
```

Let us test again:

```
mySquareRootNew[2, 5]
```

```
1.41421
```

It is interesting that if I want to call the <iterFun> function on a particular number (which is a current approximation to the square root), I need a rather unusual syntax:

```
iterFun[2][1.4]
```

```
1.41429
```

It is actually not difficult do understand: iterFun[2] gives you a pure function

```
iterFun[2]
```

$$\frac{1}{2}\left(\#1+\frac{2}{\#1}\right)\&$$

So, think of this composite (normal) expression as of a function head.What is really nice is that *Mathematica*’ssyntax allows such expressions.

To complete the story: it is not necessary for < iterFun > to return a pure function. We can define it also through SubValues, and this allows us to add an argument - check :

```
Clear[iterFunSV];
iterFunSV[number_ ? NumericQ][x_ ? NumericQ] := (x + number / x) / 2;
```

It is easy to check that this function also works when we use it in Nest.

Keep in mind that Functions < iterFunSV > and < iterFun > differ in certain subtle aspects. Unimportant here, they may become important in different circumstances. For instance, < iterFun > called on a specific number returns a pure function with this number embedded in it once and for all. We can then keep this specific one in a variable and use (call on some arguments) any number of times (this is a simple example of what is called a closure), for instance:

```
fn = iterFun[5]
```

$$\frac{1}{2}\left(\#1+\frac{5}{\#1}\right)\ \&$$

```
Map[fn, Range[10]]
```

$$\left\{3, \frac{9}{4}, \frac{7}{3}, \frac{21}{8}, 3, \frac{41}{12}, \frac{27}{7}, \frac{69}{16}, \frac{43}{9}, \frac{21}{4}\right\}$$

In contrast, < iterFunSV > can not be called with only the first "argument" (a number to embed) - it needs both arguments at the same time:

```
{iterFunSV[5], iterFunSV[5][3]}
```

$$\left\{\text{iterFunSV}[5], \frac{7}{3}\right\}$$

In some cases this may be inefficient, but on the other hand, as we saw, we can use patterns for more detailed type checks. The bottom line: these functions are different.

```
Clear[result, reslist, mySquareRoot, mySquareRootNew, iterFun];
```

- 5.4.1.6 Example: generating Hermite polynomials

Here we will generate the n-th Hermite polynomial using the Rodriguez's formula:

```
HoldForm[(-1)^n Exp[x^2] D[Exp[-x^2], {x, n}]] // TraditionalForm
```

$$(-1)^n e^{x^2} \frac{\partial^n e^{-x^2}}{\partial x^n}$$

Here is the code:

```
Clear[ourHermiteH];
ourHermiteH[n_Integer, x_] :=
 Expand[(-1)^n*Exp[x^2]*Nest[D[#, x] &, Exp[-x^2], n]]
```

For the sake of example we ignored that the built-in D can take also higher-order derivatives. Here are a few first polynomials:

```
ourHermiteH[#, x] & /@ Range[0, 3]
```

$$\{1, 2x, -2+4x^2, -12x+8x^3\}$$

We check with the built-in ones:

```
HermiteH[#, x] & /@ Range[0, 3]
```

$$\{1, 2x, -2+4x^2, -12x+8x^3\}$$

If we need a long list of polynomials, it would be more efficient to use NestList. And in this case, the use of Nest (NestList) is justified even though there exists a built-in D[expr,{x,n}] which takes higher derivatives.

```
Clear[ourHermiteList];
ourHermiteList[n_Integer, x_] :=
 Expand[(-1)^n*Exp[x^2]*NestList[D[#, x] &, Exp[-x^2], n]]
```

Check:

```
ourHermiteList[5, x]
```

$\{-1, 2x, 2-4x^2, -12x+8x^3, -12+48x^2-16x^4, 120x-160x^3+32x^5\}$

We can check how much we win by using NestList. This is the version using capabilities of D to take higher derivatives (we produce first 25 polynomials)

```
Expand[(-1)^#*Exp[x^2]*D[Exp[-x^2], {x, #}]] & /@ Range[0, 25] //
 myTiming
0.012
```

This is the same using our version with NestList

```
ourHermiteList[25, x] // myTiming
0.00681
```

We get a speed-up of about factor of 2, which is substantial.

```
Clear[ourHermiteH, ourHermiteList];
```

- 5.4.1.7 Case study: Sorting a list of numbers

The problem

Let us start with a list of numbers:

```
testlist = Table[Random[Integer, {2, 10}], {10}]

{3, 8, 4, 7, 7, 7, 4, 3, 8, 9}
```

We would like now to sort this list in the decreasing order according to the following algorithm: at any given time, we maintain a list with two sublists: the first (initially empty) gives the numbers that are already sorted, the second (initially coinciding with the original list) contains the numbers not yet sorted. A single iteration consists of finding a maximal number in the unsorted part, deleting it from there and appending it to the list of sorted numbers. The number of iterations needed to sort a list is obviously equal to the length of the list.

The sketch of the solution

Here is a function which realizes a single iteration:

```
Clear[iterSort];
iterSort[{sorted_List, unsorted_List}] :=
 Module[{max = Max[unsorted], pos},
  pos = Position[unsorted, max, 1, 1];
  {Append[sorted, max], Delete[unsorted, pos]}]
```

The code is more or less self-explanatory. We use several built-in functions, such as Max, Position, Append, Delete. Let us use it now on our test list. This is how it looks at some intermediate sorting step:

```
Nest[iterSort, {{}, testlist}, 3]
{{9, 8, 8}, {3, 4, 7, 7, 7, 4, 3}}
```

To sort the list completely:

```
Nest[iterSort, {{}, testlist}, Length[testlist]]
{{9, 8, 8, 7, 7, 7, 4, 4, 3, 3}, {}}
```

Possible bugs and automatic rule reordering

It is amusing to see what happens if we by mistake use one (or more) extra iteration

```
Nest[iterSort, {{}, testlist}, Length[testlist] +1]
{{9, 8, 8, 7, 7, 7, 4, 4, 3, 3, -∞}, {}}
```

This is due to the following behavior (or convention):

```
Max[{}]
-∞
```

If we want to be on the safe side, we will add one more definition to our function <iterSort>:

```
Clear[iterSort];
iterSort[{sorted_List, unsorted_List}] :=
  Module[{max = Max[unsorted], pos},
   pos = Position[unsorted, max, 1, 1];
   {Append[sorted, max], Delete[unsorted, pos]}];

iterSort[{sorted_List, {}}] := {sorted, {}};
```

This last definition is supposed to return back the list unchanged, once the <unsorted> part is empty. Also, because it is more specific than the first, we expect *Mathematica* to attempt to use it before it attempts to use the more general one (this is a standard rule of *Mathematica* pattern-matcher, see sections 1.2.8, 4.7.2, 4.7.3). Well, in this case we expect too much. Let us test the new function:

```
Nest[iterSort, {{}, testlist}, Length[testlist] +1]
{{9, 8, 8, 7, 7, 7, 4, 4, 3, 3, -∞}, {}}
```

It does not seem to work. To see what happens, let us look at the new definition of iterSort:

```
? iterSort
```

```
Global`iterSort

iterSort[{sorted_List, unsorted_List}] := Module[{max = Max[unsorted], pos},
  pos = Position[unsorted, max, 1, 1]; {Append[sorted, max], Delete[unsorted, pos]}]

iterSort[{sorted_List, {}}] := {sorted, {}}
```

We now see the reason: the newly added rule is placed after the main definition, and thus, has no chance to apply. But this behavior contradicts our expectations! As we know (section 1.2.8), the more specific rules are always placed by the *Mathematica* pattern-matcher before the more general ones, *when it can determine it*. By more specific I mean the rule whose pattern is completely contained in another (more general) rule's pattern as a special case.

For us it is obvious that the pattern {sorted_List,{}} represents a specific case of {sorted_List, unsorted_-List}. But not so for *Mathematica*! This kind of situations often result in some quite subtle bugs in the programs that use functions with multiple definitions. Of course, we may blame the system, but it will be more useful to understand why this happened. The point is that the way *Mathematica*'s pattern-matcher determines which rule is more specific, is completely syntax-based, rather than semantics-based. The pattern {} is syntactically different from <unsorted_List>, and determining that one is a special case of the other is already a semantic operation. Here is what we had to add instead, had we wished *Mathematica* to understand it:
iterSort[{sorted_List,unsorted_List}]/;unsorted==={}:={sorted,{}}.

Let us check:

```
Clear[iterSort];
iterSort[{sorted_List, unsorted_List}] :=
  Module[{max = Max[unsorted], pos},
   pos = Position[unsorted, max, 1, 1];
   {Append[sorted, max], Delete[unsorted, pos]}];
iterSort[{sorted_List, unsorted_List}] /; unsorted === {} :=
  {sorted, {}};
```

Check now:

```
?iterSort
Global`iterSort

iterSort[{sorted_List, unsorted_List}] /; unsorted === {} := {sorted, {}}

iterSort[{sorted_List, unsorted_List}] := Module[{max = Max[unsorted], pos},
  pos = Position[unsorted, max, 1, 1]; {Append[sorted, max], Delete[unsorted, pos]}]
```

We see that the rules have been interchanged. Of course, on the practical side, to be completely sure one can just enter the rules in the right order from the very beginning, but it is important to also understand what is going on behind the scenes. Let us check our final variant now:

```
Nest[iterSort, {{}, testlist}, Length[testlist] +1]
{{9, 8, 8, 7, 7, 7, 4, 4, 3, 3}, {}}
```

Now everything works.

Use NestList to see intermediate steps

The existence of the NestList command allows us to see all of the intermediate steps of our sorting algorithm without any extra cost - just change Nest to NestList:

```
NestList[iterSort, {{}, testlist}, Length[testlist] +1]
```

```
{{{}, {3, 8, 4, 7, 7, 7, 4, 3, 8, 9}}, {{9}, {3, 8, 4, 7, 7, 7, 4, 3, 8}},
 {{9, 8}, {3, 4, 7, 7, 7, 4, 3, 8}}, {{9, 8, 8}, {3, 4, 7, 7, 7, 4, 3}},
 {{9, 8, 8, 7}, {3, 4, 7, 7, 4, 3}}, {{9, 8, 8, 7, 7}, {3, 4, 7, 4, 3}},
 {{9, 8, 8, 7, 7, 7}, {3, 4, 4, 3}}, {{9, 8, 8, 7, 7, 7, 4}, {3, 4, 3}},
 {{9, 8, 8, 7, 7, 7, 4, 4}, {3, 3}}, {{9, 8, 8, 7, 7, 7, 4, 4, 3}, {3}},
 {{9, 8, 8, 7, 7, 7, 4, 4, 3, 3}, {}}, {{9, 8, 8, 7, 7, 7, 4, 4, 3, 3}, {}}}
```

This capability is often quite handy, in particular for debugging programs which use Nest.

Final solution

Finally, let us package our entire sort procedure into a function: first, here is our <iterSort> function once again:

```
Clear[iterSort];
iterSort[{sorted_List, unsorted_List}] :=
  Module[{max = Max[unsorted], pos},
   pos = Position[unsorted, max, 1, 1];
   {Append[sorted, max], Delete[unsorted, pos]}];

iterSort[{sorted_List, unsorted_List}] /; unsorted === {} :=
  {sorted, {}};
```

Now, the sorting function:

```
Clear[ourSort];
ourSort[sortme_List] :=
  First[Nest[iterSort, {{}, sortme}, Length[sortme]]];
```

Test:

```
Range[10]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
ourSort[Range[10]]
```

```
{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

```
Clear[ourSort, iterSort, testlist];
```

5.4.2 NestWhile and NestWhileList

These commands are used to organize a <While> loop around the Nest command. Basically, they are used when Nest is appropriate but we don't know in advance how many iterations are needed. The format of the command in the simplest form is :

```
NestWhile[function, expr, test]
```

So, the last argument of Nest is replaced by the argument <test> here. The argument <test> has to be a function (pure or pattern-defined), which applies to the result of the last iteration, and gives True or False (i.e., a predicate). Once it no longer gives True (notice that this is not the same as giving explicit False), the loop stops. Simple examples:

5.4.2.1 Simple examples

Deleting numbers from the list

Here is a list containing in general zeros, positive and negative integers.

```
Clear[testlist];
testlist = Table[Random[Integer, {-10, 10}], {15}]
{3, 2, -9, 0, 5, 3, -6, -1, -7, 0, -3, 0, -8, -7, -1}
```

This will drop the first element in the list repeatedly until it meets a first negative number:

```
NestWhile[Drop[#, 1] &, testlist, NonNegative[First[#]] &]
{-9, 0, 5, 3, -6, -1, -7, 0, -3, 0, -8, -7, -1}
```

Warning : efficiency pitfall

Note that this method in fact contains a rather unobvious efficiency pitfall which will cause problems for large lists. We already discussed that it is inefficient to use Append and Prepend in creation of lists. This was so because at every stage a whole list was copied to append a single element. But let us recall that most functions in *Mathematica* work without side effects, which means that they create a copy and operate on this copy. Here we drop element by element, rather than append, and thus our first reaction is that things are fine (really, the size of the list does not have to be increased). But this does not matter. What matters is that Drop creates a copy of the list just as Append, and is no better in this sense. To illustrate, consider deleting elements in a loop one by one. We will create a test function and measure timings for various list sizes: 10,100,1000,10000 and 50000 elements.

```
Clear[testFun];
testFun[n_Integer] :=
 Module[{m = 1}, NestWhile[Drop[#, 1] &, Range[n], m ++ < n &]]
```

Check :

```
Map[myTiming[testFun[#]] &, {10, 100, 1000, 10 000, 50 000}]
{0.00017, 0.0016, 0.0211, 0.3, 5.168}
```

The first several numbers look as if the timing was linear as a function of the list size, and this would contradict our guess above, but this is an illusion. This simply means that for small lists, copying lists is very efficient and the main time is spent on incrementing < m > and checking the terminating condition. Indeed, this version does not involve NestWhile and condition checks:

```
Clear[testFun1];
testFun1[n_Integer] := Module[{start = Range[n]},
   Do[start = Drop[start, 1], {n}];
   start];
```

Check :

```
Map[myTiming[testFun1[#]] &, {10, 100, 1000, 10 000, 50 000}]
{0.00015, 0.000471, 0.01, 0.19, 4.216}
```

This reveals that the timing is not really linear in the list size even for smaller lists, although it is close to linear up to rather large list sizes (1000), and even better than linear for small lists. But in any case, for larger lists the timings confirm our guess above. Using a built-in NestWhile does not change the fact that the copy of the list is created at every iteration - this is a property the function being nested (Drop in this case).

The bottom line: avoid modifying large lists in place many times by small changes like deleting or appending a single element at a time. Also, remember that most built-in functions work without side effects and this means that they *necessarily* make copies of objects passed to them.

Imitating FromDigits

We are given a number, say 7423. We want to split it into a list of digits (this is done by the built-in FromDigits command). Here is the almost complete code:

```
NestWhile[
 {Prepend[#[[1]], Mod[#[[2]], 10]], IntegerPart[#[[2]] / 10]} &,
 {{}, 7423}, #[[2]] ≠ 0 &]
{{7, 4, 2, 3}, 0}
```

We see, that it remains to take the first part of the list. To see, what is going on, it would be handy to see the intermediate steps. Here we recall that NestWhileList, which is related to NestWhile in the same way as NestList is related to Nest, gives all intermediate results in a list. So:

```
NestWhileList[
 {Prepend[#[[1]], Mod[#[[2]], 10]], IntegerPart[#[[2]] / 10]} &,
 {{}, 7423}, #[[2]] ≠ 0 &]
{{{}, 7423}, {{3}, 742}, {{2, 3}, 74}, {{4, 2, 3}, 7}, {{7, 4, 2, 3}, 0}}
```

So, we start with an empty first sublist and a number. Then, we place the remainders of division by 10 (i.e., digits) to the left sublist (notice the use of the Prepend command. Should we use append here, and the numbers would be in reverse order), while replacing the number by an integer part of itself divided by 10. The loop stops when this integer part becomes zero. This procedure can be trivially generalized to any base. So, our function would be

```
Clear[ourFromDigits];
ourFromDigits[num_Integer, base_Integer] :=
  First[NestWhile[{Prepend[#[[1]], Mod[#[[2]], base]],
      IntegerPart[#[[2]] / base]} &, {{}, num}, #[[2]] ≠ 0 &]];
```

Check:

```
ourFromDigits[10, 2]
{1, 0, 1, 0}
ourFromDigits[120, 10]
{1, 2, 0}
ourFromDigits[120, 2]
{1, 1, 1, 1, 0, 0, 0}
Clear[ourFromDigits];
```

- 5.4.2.2 More general uses of NestWhile

There exist more complicated forms of NestWhile(List), which take as arguments for the test condition at most the last <m> results. The syntax is

```
NestWhile[function, expr, test, m]
```

This is potentially a very powerful capability. Let us now give a few more examples, some of which will fully explore this general form.

- 5.4.2.3 Example: restricted random sequences

Suppose we want to generate random integers in the range {1,10} and stop when the sum of the last 3 generated numbers exceeds some number, say 20. Here is the code:

```
NestWhileList[Random[Integer, {1, 10}] &, 0, Plus[##] < 20 &, 3]
{0, 4, 3, 3, 6, 3, 4, 7, 2, 10, 2, 8}
```

Warning : a tricky bug

Note that here we used the SlotSequence (##) (section 4.11.1.8). Had we used the usual slot < # > (by mistake), and only the first of the three numbers would be used in Plus. Here I construct an example which explicitly shows this behavior:

```
Module[{n = 1, lst = Range[15, 25]},
 NestWhileList[lst[[n ++]] &, 0, Plus[#] < 20 &, 3]]
{0, 15, 16, 17, 18, 19, 20, 21, 22}
```

What is really important is that no error was generated in this case, due to the way pure functions treat excessive variables passed to them (they silently ignore them, see section 4.11.1.6). This sorts of bugs are hard to catch.

- 5.4.2.4 Example: visualizing poker probabilities

Consider a simplified version of poker where we are interested in ranks of the cards, but not suits (this will then exclude certain combinations). These are the ranks (or cards):

```
cards = {2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A};
```

We want to randomly deal the cards, until certain combinations occur - then we stop. The probability of each combination can be related to the average length of the generated sequence of cards (if we generate many sequences). Let us first write a function which will deal a random card:

```
Clear[randomCard];
randomCard[_] := cards[[Random[Integer, {1, 13}]]];
```

This definition seems fine, but it has two drawbacks. To see them, let us look at the resulting global definition of this function:

```
? randomCard
```

```
Global`randomCard

randomCard[_] := cards⟦Random[Integer, {1, 13}]⟧
```

We see that it contains <cards>. Thus, the first flaw is that the list <cards> will be recomputed every time the function is called. The second flaw is even more important - we made a function implicitly depend on a global variable <cards>. This is a pretty bad habit. It would be nicer if we could embed the current value of the list <cards> straight into the function definition. Here is the code which does it:

```
Clear[randomCard];
With[{ourcards = cards},
  randomCard[_] := ourcards[[Random[Integer, {1, 13}]]]];
```

We will cover this technique in full generality later. For now, just observe the result:

```
? randomCard
```

```
Global`randomCard

randomCard[_] := {2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A}⟦Random[Integer, {1, 13}]⟧
```

Now we are fine - the function definition is now completely independent of the global variable <cards> and thus insensitive to possible future changes of this variable.

Let us now check the function. For instance,

```
randomCard /@ Range[5]
{2, 9, A, 3, 3}
```

Now, let us say we are interested in only pairs, three-of -a-kind and two-pair combinations for the time being. The way we will solve the problem is first to construct the patterns for these combinations:

```
Clear[pairPattern, twoPairPattern, threePattern, a, b, c, d];
```

```
pairPattern = {a_, a_, b_, c_, d_} |
   {b_, a_, a_, c_, d_} | {b_, c_, a_, a_, d_} | {b_, c_, d_, a_, a_};
twoPairPattern =
  {a_, a_, b_, b_, c_} | {c_, a_, a_, b_, b_} | {a_, a_, c_, b_, b_};
threePattern =
  {a_, a_, a_, b_, c_} | {b_, a_, a_, a_, c_} | {b_, c_, a_, a_, a_};
```

The reason that we need only those alternatives used above is that we are going to Sort the hands of cards, and for sorted hands these alternatives exhaust all the possibilities.

Now, consider, for instance, pairs. Here is the code to generate the sequence of cards.

```
NestWhileList[randomCard, {},
 Not[MatchQ[Sort[{##}], pairPattern]] &, 5]
```

```
{{}, K, 10, J, 7, 5, J}
```

Notice the way that the condition testing function is written. It takes 5 arguments, which is, 5 most recently generated cards. To avoid writing explicitly every argument, we use SlotSequence (##) (section 4.11.1.8). We combine the cards into a list, and then Sort it. Then, the sorted hand of cards is compared to the pattern by MatchQ. If the pattern matches, the loop terminates. You can run it several times to get different sequences.

The next step will be to write a function that takes the card pattern and generates the sequence of cards.

```
Clear[generateCardSequence];
generateCardSequence[cardpattern_] := Rest[NestWhileList[
    randomCard, {}, Not[MatchQ[Sort[{##}], cardpattern]] &, 5]];
```

We dropped the first element which is an empty list. Now we can check:

```
generateCardSequence[pairPattern]
```

```
{7, 8, 4, A, Q, 7, 4}
```

```
generateCardSequence[twoPairPattern]
```

```
{J, A, 10, Q, 3, J, 4, 10, Q, 6, 6, 6, 8, 3, A, 3,
 4, 5, 3, 3, 3, 10, K, 6, 8, 4, 6, 9, 5, 2, 3, 4, 5, 4, 5}
```

```
generateCardSequence[threePattern]
```

```
{8, Q, Q, 10, 3, 6, 8, 9, J, 3, 7, 2, 5, 8, A, J, 6, 4, A,
 Q, A, Q, K, 7, A, 5, J, 7, A, 3, 3, 6, 10, 2, A, 3, A, Q, 7,
 9, 7, K, 6, K, 8, A, 2, 7, 5, 3, Q, 6, 4, Q, 10, 10, Q, 9, K,
 K, 10, 4, 3, K, 10, J, 8, 3, A, 10, Q, 4, 7, 2, 9, 2, K, 2}
```

It is interesting that a pattern can also serve as an argument of the function. This idea may seem somewhat unusual since usually patterns are used to define functions (formal parameters), but not as actual arguments passed to them. The pattern in the definition of <generateCardSequence> means "any single expression", in particular it may be another pattern.

The probability of the occurrence of a combination can be estimated by 5/<average length of the generated sequence> (one could alternatively average 5/length_i over <i>). We can define a function which will generate the given number of sequences for a given pattern and compute this quantity:

First, define an auxiliary function listAverage:

```
Clear[listAverage];
listAverage[x_List] := N[Total[x] / Length[x]];
```

Now the main function:

```
Clear[probEstimate];
probEstimate[pattern_, numseqs_Integer ? Positive] :=
  5 / listAverage[
    Table[Length[generateCardSequence[pattern]], {numseqs}]];
```

Let us see. For a pair:

```
probEstimate[pairPattern, 400]
0.827815
```

For a three-of-a-kind:

```
probEstimate[threePattern, 400]
0.141293
```

For two pairs:

```
probEstimate[twoPairPattern, 400]
0.229938
```

One has to keep in mind that for smaller probabilities, one has to consider larger number of sequences, to get a representative sample. Also, smaller probabilities mean that in general each sequence will be longer (it will take on the average more trials to produce a less probable combination). These two circumstances combined together mean that, as we go to higher-ranked combinations, our method will quickly become inefficient.

Our implementation is perhaps not the fastest one, and probably also not the slowest one. The pattern-matching here is purely syntactic and thus should be efficient enough. What I wanted to illustrate here is the use of NestWhileList and patterns in a rather non-trivial setting, so that one can see that these two seemingly disjoint programming styles may nicely coexist and complement each other.

We leave it as an exercise to the reader to create patterns for other combinations, and get estimates of their probabilities (taking into account efficiency considerations discussed above, many of the combinations will be in practice beyond the reach of the present method).

```
Clear[probEstimate, listAverage, generateCardSequence,
  pairPattern, twoPairPattern, threePattern, randomCard];
```

- 5.4.2.5 Example: generating distinct random numbers

Here is the problem: we need to generate a given number of random integers, in the specified range, but such that all generated numbers are different. The idea of the present solution would be to generate a number, then check if it is already in the list, and if so - disregard it. If not, place it in the list, until the total number of integers generated will match the requested quantity. Let me immediately comment that such an algorithm is quite inefficient, but we will improve it along the way.

To solve this problem, we will reformulate it somewhat: instead of checking the presence of the number and then decide whether or not it has to be included, we will always add it to the least, but then take a Union operation which will eliminate redundant elements. Thus, the loop termination condition will be that the length of the resulting list is equal to a requested quantity of random integers. Since the standard Union operation sorts the numbers, we will need an alternative one which does not. Its form and use is illustrated below (we use our implementation of this function developed in section 5.2.6.2.5). A good alternative implementation can be obtained with the help of Reap and Sow commands - it is given as an example in *Mathematica* Help).

Here is our test list:

```
Clear[testlist];
testlist = Table[Random[Integer, {1, 10}], {15}]
{5, 6, 7, 10, 4, 9, 3, 2, 5, 4, 4, 4, 3, 1, 10}
```

This is the implementation:

```
Clear[unsortedUnion];
unsortedUnion[x_List] :=
  Extract[x, Sort[Union[x] /. Dispatch[MapIndexed[Rule, x]]]];
```

Observe:

```
unsortedUnion[testlist]
{5, 6, 7, 10, 4, 9, 3, 2, 1}
```

Now, this is the code that solves the problem:

```
Clear[randomNums];
randomNums[numrange_List, n_Integer] :=
 NestWhile[unsortedUnion[Append[#, Random[Integer, numrange]]] &,
  {}, Length[#] < n &]
```

The code is self-explanatory. Let us test the function. For instance:

```
randomNums[{1, 15}, 10]
{10, 4, 9, 6, 5, 7, 2, 15, 14, 12}
randomNums[{1, 1000}, 20]
{780, 615, 505, 596, 51, 990, 791, 90, 347,
 427, 321, 891, 419, 219, 823, 858, 163, 821, 353, 467}
```

This algorithm may be made much more efficient if we append more than one random number at a time, so that we don’ t call Append and UnsortedUnion for every single number (since their use is the biggest bottleneck). If the list at the end contains more numbers than needed, the extra ones can be dropped.

```
Clear[randomNumsBetter];
randomNumsBetter[numrange_List,
  n_Integer, updatenum_Integer: 100] := Take[
  NestWhile[unsortedUnion[Join[#, Table[Random[Integer, numrange],
       {updatenum}]]] &, {}, Length[#] < n &], n]
```

Note the use of optional pattern <updatenum_Integer: 100> to implement the defualt value for the <update number> argument. Let us compare the performance :

```
randomNums[{1, 1000}, 300] // Short // Timing
{0.53, {783, 57, 383, <<294>>, 314, 927, 928}}
randomNumsBetter[{1, 1000}, 300] // Short // Timing
{0.02, {905, 514, 457, <<294>>, 35, 936, 358}}
```

By tuning the number of random integers generated at once in a single iteration, one can further improve the performance :

```
randomNumsBetter[{1, 1000}, 300, 500] // Short // Timing
{0.01, {548, 230, 977, <<294>>, 512, 45, 232}}
```

This is already more or less acceptable (speed - up 100 times w.r.t. naive version for the above parameters on my machine).

If all that matters is a set of numbers but not the order in which they follow, one can further speed - up our function by replacing UnsortedUnion by Union.

```
Clear[randomNumsOrdered];
randomNumsOrdered[numrange_List, n_Integer,
   updatenum_Integer: 100] := With[{range = numrange},
   Take[NestWhile[Union[Join[#, Array[Random[Integer, range] &,
         {updatenum}]]] &, {}, Length[#] < n &], n]];
```

Check :

```
randomNumsOrdered[{1, 1000}, 300, 500] // Short // Timing
```

$\{2.20102\times 10^{-14}$, {2, 3, 7, 8, <<293>>, 757, 758, 759}$\}$

Here we make a power test by generating 30000 random numbers in the range 1 .. 1000000, and parameter <updatenum> tuned to 7000 (roughly a quarter of the total number of integers needed).

```
randomNumsOrdered[{1, 1 000 000}, 30 000, 7000] // Short // Timing
{0.091, {70, 100, ≪29 996≫, 870 393, 870 399}}
```

Of course, a good question to ask would be what is the resulting distribution (probability density) for the random numbers obtained in this way, and whether it is what we want, but that's another question.

```
Clear[testlist, randomNums, UnsortedUnion]
```

- 5.4.2.6 Example: the Collatz problem

Since the previous example may have left the reader with an impression that NestWhile is only good to produce inefficient solutions, we will now consider an example where it is perhaps the most appropriate command to use, both in the sense of elegance and efficiency.

The Collatz iteration is given by:

```
Clear[c];
c[n_ ? OddQ] := 3 * n + 1;
c[n_ ? EvenQ] := n / 2;
```

It has been noticed that, regardless of the starting number <n>, the sequence of numbers that result from the repeated application of the function <c> will always eventually go to 1 (although this has not been rigorously proven). The most interesting question is how the length of the Collatz sequence depends on the starting number.

We will be interested in implementing the Collatz sequence. First, consider the implementation from the "Computer science in *Mathematica*" by Roman Maeder.

```
Clear[collatzSequence];
collatzSequence[1] = {1};
collatzSequence[n_] := Prepend[collatzSequence[c[n]], n];
```

Look carefully at this implementation. The idea behind is beautiful: we recursively define the sequence by prepending a starting number to the sequence which starts with the transformed starting number. The separate base case guarantees a proper termination of the recursion.

For example:

```
collatzSequence[99]
{99, 298, 149, 448, 224, 112, 56, 28, 14, 7, 22,
 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

Let us test the performance of this solution (I chose the powers of 2 since then the length of the Collatz sequence is known in advance and is equal to the power of 2)

```
Block[{$RecursionLimit = Infinity},
 Print[Timing[collatzSequence[2^99];],
   Timing[collatzSequence[2^999];],
   Timing[collatzSequence[2^2000];], Timing[
    collatzSequence[2^5000];], Timing[collatzSequence[2^9999];]];]
```

```
{0.01 Second, Null}{0.05 Second, Null}{0.14 Second, Null}{1.102 Second, Null}{11.226 Second, Null}
```

We had to temporarily disable a limit on number of recursive calls (recursion depth) since we will need the depth of recursion equal to the power of 2, in each case. The standard limit is 256. <Block> is used to make this modification local to its interior. We use <Block> when we want some function or expression to temporarily "forget" the associated external (global) rules.

The inefficiency is (c.f. Wagner'96) due to modifications of large lists in place at any iteration stage. This is necessary in this method, since the length of the sequence is not known an advance. The complexity of the program should be roughly proportional to N^3/2, where N is the length of the Collatz sequence.

Here is an alternative implementation using NestWhileList:

```
Clear[colSequence];
colSequence[q_Integer] := NestWhileList[c, q, #1 ≠ 1 &];
```

Check:

```
colSequence[99]
```

```
{99, 298, 149, 448, 224, 112, 56, 28, 14, 7, 22,
 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

We now test the performance :

```
Block[{$RecursionLimit = Infinity},
 Print[Timing[colSequence[2^99];],
   Timing[colSequence[2^999];], Timing[colSequence[2^2000];],
   Timing[colSequence[2^5000];], Timing[colSequence[2^9999];]];]
```

```
{0.01, Null}{0.01, Null}{0.05, Null}{0.1, Null}{0.211, Null}
```

This version does not communicate the idea and recursive nature of the Collatz sequence so clearly (which was probably the main motivation of Maeder. Besides, NestWhileList did not exist at the time), but the performance of this version is much better. This is because, the sequence (list) is created internally inside NestWhileList, and we don't have to modify large lists in place. The complexity of this program depends on details of internal implementation of NestWhileList, but could be even linear or log-linear, if c[x] is approximately constant-time (or log). We see that this problem is tailor-made for NestWhileList. It can be also seen by the conciseness of the code. Note that should we have only NestWhile at our disposal, this solution would not be possible - in this case we needed exactly NestWhileList.

Generally, many problems involving building large lists element by element and when the next element depends on the previous element(s), can be reformulated such that they can be solved by NestWhileList. This is advantageous in *Mathematica* programming, because one can think of NestWhileList as an efficient cousin of the standard procedural loops (which are usually inefficient in *Mathematica*). In the next case study of the Fibonacci numbers we will further dwell on this topic.

```
Clear[c, collatzSequence, colSequence];
```

■ 5.4.2.7 Case study: on automatic and programmatic construction of patterns - patterns for poker combinations revisited (not NestWhile - related)

The problem

Many problems admit in *Mathematica* elegant and efficient solutions based on patterns and pattern - matching. But often it may be desirable to also create the patterns programmatically, especially when a pattern is combined from a large number of alternative patterns. We will illustrate the possibility of programmatic pattern construction on the just considered example of poker combinations. Here are the patterns we had :

```
Clear[pairPattern, twoPairPattern, threePattern, a, b, c, d];

pairPattern = {a_, a_, b_, c_, d_} |
   {b_, a_, a_, c_, d_} | {b_, c_, a_, a_, d_} | {b_, c_, d_, a_, a_};

twoPairPattern =
  {a_, a_, b_, b_, c_} | {c_, a_, a_, b_, b_} | {a_, a_, c_, b_, b_};

threePattern =
  {a_, a_, a_, b_, c_} | {b_, a_, a_, a_, c_} | {b_, c_, a_, a_, a_};
```

First level of automation

Essentially the same patterns can be created in a more automatic fashion :

```
Alternatives @@
 Map[RotateRight[{a_, a_, b_, c_, d_}, #] &, Range[0, 3]]

{a_, a_, b_, c_, d_} | {d_, a_, a_, b_, c_} |
 {c_, d_, a_, a_, b_} | {b_, c_, d_, a_, a_}

Alternatives @@
 Map[RotateRight[{a_, a_, a_, c_, d_}, #] &, Range[0, 2]]

{a_, a_, a_, c_, d_} | {d_, a_, a_, a_, c_} | {c_, d_, a_, a_, a_}

Alternatives @@
 Map[RotateRight[{a_, a_, b_, b_, c_}, #] &, Range[1, 5, 2]]

{c_, a_, a_, b_, b_} | {b_, b_, c_, a_, a_} | {a_, a_, b_, b_, c_}
```

Second level of automation : constructing patterns completely programmatically

If desired, one can achieve an even higher level of automation. Consider the following function (which we will cover later in detail), which gives all distinct partitions for a given integer:

```
Clear[distinctPartitions];
distinctPartitions[n_Integer] := Block[{fn},
   fn[x_List, 0] := x;
  fn[x_List, num_Integer] := Map[fn[Flatten[{x, #}], num -#] &,
    Range[num, If[x === {}, 1, x[[-1]]], -1]];
  Sort[Cases[fn[{}, n], {__Integer}, Infinity],
   Length[#1] < Length[#2] &]]
```

Let us realize that the following partitions of 5 :

```
Rest[distinctPartitions[5]]
{{1, 4}, {2, 3}, {1, 1, 3}, {1, 2, 2}, {1, 1, 1, 2}, {1, 1, 1, 1, 1}}
```

realize poker combinations four-of-a-kind, full house, three - of - a- kind, two pairs, pair and just highest rank card. The maximum number of distinct variables for patterns is clearly 5. Thus, we create 5 dummy variables:

```
vars = Table[Unique[], {5}]
{$11, $12, $13, $14, $15}
```

The function below will take a given partition, and a list of variables. It will return a combination of variables where the multiplicity of each distinct variable corresponds to one of the numbers in the partition.

```
Clear[patternVars];
patternVars[partition_List, vars_List] :=
  Flatten[Table[vars[[#]], {partition[[#]]}] & /@
    Range[Length[partition]]];
```

For example :

```
patternVars[{1, 1, 3}, vars]
{$11, $12, $13, $13, $13}
```

To create all pattern sequences at once, we will simply use Permutations :

```
patternVars[#, vars] & /@ Permutations[{1, 1, 3}]
{{$11, $12, $13, $13, $13},
 {$11, $12, $12, $12, $13}, {$11, $11, $11, $12, $13}}
```

We see that the actual variable names are always different for the 3 identical cards, but for us now this is not a problem. If in some case it is, one can rewrite functions appropriately to take this into account, so that the variable names are also permuted accordingly.

The final thing is to convert this to real pattern, which we do with the code :

```
Clear[makePattern];
makePattern[varcomb_List] :=
  With[{varc = varcomb}, Thread[Pattern[varc, Blank[]]]];
```

The With construct was used to avoid the error message which appears when the r.h.s. of an assignment contains the same pattern as used in the l.h.s. This situation looks so to *Mathematica* interpreter. Now check :

```
makePattern[patternVars[{1, 1, 3}, vars]]
{$11_, $12_, $13_, $13_, $13_}
```

Now all we have to do is to Map our makePattern on the prepared list of variable combinations :

```
Map[makePattern[patternVars[#, vars]] &, Permutations[{1, 1, 3}]]
{{$11_, $12_, $13_, $13_, $13_},
 {$11_, $12_, $12_, $12_, $13_}, {$11_, $11_, $11_, $12_, $13_}}
```

And finally Apply < Alternatives > :

```
Alternatives @@
 Map[makePattern[patternVars[#, vars]] &, Permutations[{1, 1, 3}]]
{$11_, $12_, $13_, $13_, $13_} |
 {$11_, $12_, $12_, $12_, $13_} | {$11_, $11_, $11_, $12_, $13_}
```

So, our second function will be :

```
Clear[getTotalPattern];
getTotalPattern[partition_List] := Module[{vars},
  vars = Table[Unique[], {Length[partition]}];
  Apply[If[Length[#] == 1, Identity, Alternatives], #] &@Map[
    makePattern[patternVars[#, vars]] &, Permutations[partition]]]
```

We also refined the Apply Alternatives part so that when there is a single pattern, Identity head is applied (i.e., the same pattern is returned). For example :

```
getTotalPattern[{1, 1, 3}]
{$62_, $63_, $64_, $64_, $64_} |
 {$62_, $63_, $63_, $63_, $64_} | {$62_, $62_, $62_, $63_, $64_}
```

Let us now create all the patterns for our combinations. We will just need to Map the < getTotalPattern > function on a list of our combinations (partitions of 5) :

```
Map[getTotalPattern, distinctPartitions[5]]
{{$42_, $42_, $42_, $42_, $42_},
 {$43_, $44_, $44_, $44_, $44_} | {$43_, $43_, $43_, $43_, $44_},
 {$45_, $45_, $46_, $46_, $46_} | {$45_, $45_, $45_, $46_, $46_},
 {$47_, $48_, $49_, $49_, $49_} | {$47_, $48_, $48_, $48_, $49_} |
  {$47_, $47_, $47_, $48_, $49_}, {$50_, $51_, $51_, $52_, $52_} |
  {$50_, $50_, $51_, $52_, $52_} | {$50_, $50_, $51_, $51_, $52_},
 {$53_, $54_, $55_, $56_, $56_} | {$53_, $54_, $55_, $55_, $56_} |
  {$53_, $54_, $54_, $55_, $56_} | {$53_, $53_, $54_, $55_, $56_},
 {$57_, $58_, $59_, $60_, $61_}}
```

In this way, we constructed completely programmatically patterns for many poker combinations. Let us collect all necessary functions together once again:

```
Clear[patternVars];
patternVars[partition_List, vars_List] :=
  Flatten[Table[vars[[#]], {partition[[#]]}] & /@
    Range[Length[partition]]];
```

```
Clear[makePattern];
makePattern[varcomb_List] :=
  With[{varc = varcomb}, Thread[Pattern[varc, Blank[]]]];
```

```
Clear[getTotalPattern];
getTotalPattern[partition_List] := Module[{vars},
  vars = Table[Unique[], {Length[partition]}];
  Apply[If[Length[#] == 1, Identity, Alternatives], #] &@Map[
    makePattern[patternVars[#, vars]] &, Permutations[partition]]]
```

5.4.2.8 Case study: Fibonacci numbers

The problem

Fibonacci numbers are defined as follows: fib(0)=fib(1) =1,fib(n) = fib(n-1)+fib(n-2), n≥2. This is a standard example to show that the use of recursion (which seems natural in this case) may lead to a huge overhead (exponential in this case), due to the massive redundant recomputations of the same quantities. We will be considering three somewhat different problems: produce a given Fibonacci number fib(n), produce a list of first <n> Fibonacci numbers, and produce all Fibonacci numbers less than a given number.

The standard recursive solution in Mathematica

Let me first briefly show the recursive solution to the first problem. For it, we have to transfer the definition to *Mathematica* code practically verbatim:

```
Clear[fib];
fib[0] := 1;
fib[1] := 1;
fib[n_Integer] := fib[n - 1] + fib[n - 2];
```

Here are the first few numbers:

```
fib /@ Range[0, 20]
```

```
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10 946}
```

Efficiency analysis

Let us produce the first few numbers and measure the performance in each case:

```
{fib[3], myTiming[fib[3]]}
```

```
{3, 0.00002}
```

```
{fib[5], myTiming[fib[5]]}
```

```
{8, 0.000046}
```

```
{fib[10], myTiming[fib[10]]}
```

```
{89, 0.000581}
```

```
{fib[15], myTiming[fib[15]]}
```

```
{987, 0.0101}
```

```
{fib[20], myTiming[fib[20]]}
```

```
{10 946, 0.0771}
```

It is not difficult to check that the time needed to compute n-th number grows exponentially with n. The reason for this inefficiency is that smaller Fibonacci numbers are used in the computation of all higher numbers. Essentially, the recursion builds up a binary tree of function calls, of depth n, and each given m-th number populates the m-th level of the tree, if we count from the bottom. Thus, smaller Fibonacci numbers have to be computed exponentially many times (2^(n-m), very roughly).

To display a function call tree, we use two mutually recursive functions fib1 and fib2. Basically, fib2 is just a wrapper to keep track of the function calls:

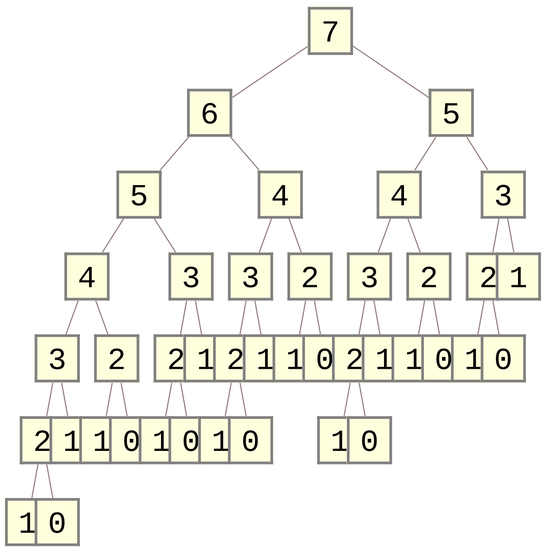
```
Clear[fib1, fib2];
fib1[0] = fib2[0, {}];
fib1[1] = fib2[1, {}];
fib1[n_Integer] /; n ≥ 2 := fib2[n, {fib1[n - 1], fib1[n - 2]}]
```

Now, for example, the 7-th Fibonacci number:

```
expr = fib1[7]
fib2[7, {fib2[6,
   {fib2[5, {fib2[4, {fib2[3, {fib2[2, {fib2[1, {}], fib2[0, {}]}],
          fib2[1, {}]}], fib2[2, {fib2[1, {}], fib2[0, {}]}]}],
        fib2[3, {fib2[2, {fib2[1, {}], fib2[0, {}]}], fib2[1, {}]}]}],
     fib2[4, {fib2[3, {fib2[2, {fib2[1, {}], fib2[0, {}]}],
         fib2[1, {}]}], fib2[2, {fib2[1, {}], fib2[0, {}]}]}]}],
  fib2[5, {fib2[4, {fib2[3, {fib2[2, {fib2[1, {}], fib2[0, {}]}],
         fib2[1, {}]}], fib2[2, {fib2[1, {}], fib2[0, {}]}]}],
     fib2[3, {fib2[2, {fib2[1, {}], fib2[0, {}]}], fib2[1, {}]}]}]}]
```

Here is a tree form of the above expression with extra wrappers <fib2> removed, which shows best the picture of recursive function calls

```
TreeForm[
 expr /. {List → Sequence} //. {fib2[x_, y__] :> x[y], fib2[x_] ⧴ x}]
```



We can count how many times each fib2 was called. First create a list of patterns:

```
patterns = fib2[#, __] & /@ Range[0, 7]

{fib2[0, __], fib2[1, __], fib2[2, __], fib2[3, __],
 fib2[4, __], fib2[5, __], fib2[6, __], fib2[7, __]}
```

Now Map the count function:

```
{# /. fib2[a_, __] ⧴ fib2[a], Count[{expr}, #, Infinity]} & /@ patterns
{{fib2[0], 8}, {fib2[1], 13}, {fib2[2], 8}, {fib2[3], 5},
 {fib2[4], 3}, {fib2[5], 2}, {fib2[6], 1}, {fib2[7], 1}}
```

The rule #/.fib2[a_,__]⧴fib2[a] simply removes the pattern sign from fib2[number,__]. The funny thing is that the numbers of function calls form the reversed Fibonacci sequence - we could have guessed that.

The standard iterative solution (procedural)

There exists an elegant way in *Mathematica* to get an efficient implementation for the Fibonacci numbers using dynamic programming, and we will revisit this problem once we cover this topic. For now, let us see if we can find a more efficient implementation using the functions and techniques we already know.

The first thing which comes to mind is a procedural iterative implementation:

```
Clear[fibProc];
fibProc[n_Integer] :=
  Module[{prev, prprev, i, temp}, For[prev = prprev = i = 1, i < n,
    i ++, temp = prprev; prprev = prev; prev = prev +temp;];
   prev];
```

For instance:

```
fibProc[7]
21
```

Producing a sequence of first n Fibonacci numbers

This above solution is decent if all one wants is to compute a given Fibonacci number. But what if we need a full sequence up to a given number? The better procedural solution then would be to create a list of numbers inside the For loop:

```
Clear[fibProcList];
fibProcList[n_Integer] :=
  Module[{prev, prprev, i, temp, reslist = Table[0, {n}]}, For[
    prev = prprev = reslist[[1]] = i = 1, i < n, i ++, temp = prprev;
    prprev = reslist[[i +1]] = prev; prev = prev +temp;];
   reslist];
```

For example:

```
fibProcList[30]
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
 610, 987, 1597, 2584, 4181, 6765, 10 946, 17 711, 28 657,
 46 368, 75 025, 121 393, 196 418, 317 811, 514 229, 832 040}
```

```
fibProcList[30] // myTiming
0.000521
```

The timing is quite good actually (for *Mathematica* implementation). For this particular formulation, the procedural solution is among the best. Notice that the list to store the results is pre-allocated from the beginning. Had we started with an empty list and then Append to it repeatedly, the performance would be far worse. This leads us to appreciate another rule: pre-allocate large lists. But here, this was possible because we knew from the beginning how many numbers we want.

Producing all Fibonacci numbers less than a given number

Consider now a different formulation: we want all Fibonacci numbers smaller than a given number. Here, we don't know a priori how large a list we will need. Of course, we can perform an analysis, make estimates etc, then preallocate a list guaranteed to be large enough, and then delete extra (unused) slots in the list at the end. But there could be similar problems for which such analysis is very difficult, and then one may end up pre-allocating huge lists where much smaller ones are actually needed, and thus waste resources.

On the other hand, Append-ing to a list repeatedly has efficiency problems well - known to us by now (this is not an absolute restriction however. In cases when the body of the loop - the function which produces the next result from the previous ones - is very computationally-demanding and takes most of the time, and at the same time the length of the list of results is not too large, the cost of Append-ing the list may be negligible w.r.t other operations. In such cases, using Append is certainly a possibility).

Let us see if we can find another solution. What we would like to do is to make the system create a list of results internally, but stop depending on some condition applied to the results. This sounds like a good case for NestWhileList.

What is really important here is that NestWhileList has a formulation where it has access to a given number of the most recent results. On the other hand, a limitation of Nest or NestList is that a new function output is based only on the last result - the nested function does not have access to any past results except the last one. But for the Fibonacci numbers problem, we need to know the last 2 results. So, the main idea of the solution is the following: given the use of NestWhileList in the form

NestWhileList[f,expr,test,m] ,

(m gives a maximal number of the most recent results to be supplied to the test condition) we can use a global variable to communicate the given number of most recent results from the <test> function to f:

```
Clear[fibList];
fibList[max_Integer] := Module[{a = 0},
   Drop[NestWhileList[a + # &, 1, (a = #1) < max &, 2], -2]];
```

What happens here is that the test function contains a side effect - an assignment to a variable <a>. A global variable <a> (well, we made it local with Module, but it is still "global" for the body of NestWhileList) is used to communicate to the function f (which is here a pure function that adds a "constant" <a> to the previous input), the current value of <a>, which is, in terms of the procedural solution, a value of <prprev> - the "previous previous" result. We have to Drop the last 2 results since they will be larger than the limiting number. Check:

```
fibList[1 000 000]
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
 610, 987, 1597, 2584, 4181, 6765, 10 946, 17 711, 28 657,
 46 368, 75 025, 121 393, 196 418, 317 811, 514 229, 832 040}
fibList[1 000 000] // myTiming
0.000441
```

We see that this solution not only does solve our present problem, but also beats our best procedural version for a previous fixed-n formulation! Also, the code is more concise and transparent, less variables are introduced etc. If we look back at the code, what happens is rather non-trivial: at every stage, a different value of <a> is embedded into a pure function, which is then used to produce a new number. In that sense, at each stage we change a nesting function definition.

This technique should be applicable to other situations where we have to produce a list of results with the number of them unknown beforehand, and when both every next result and the termination condition depend on the results produced already.

```
Clear[fib, fib1, fib2, fibProc, fibProcList, fibList];
```

5.5 Fold and FoldList

5.5.1 Fold: syntax and starting examples

Fold is a generalization of Nest, for the case when the nested function is a function of two variables, rather than of a single variable. While the first supplied variable is again a result of the previous action of the function itself, a second argument is supplied externally by Fold, from a list which is a parameter to Fold. So, the syntax:

```
Fold[f, start, secarglist]
```

Simple example:

```
Clear[f];
Fold[f, x, {a, b, c}]
```

```
f[f[f[x, a], b], c]
```

Fold is a remarkably useful function. One may think of it as follows: if <f> can be thought of as a function which realizes a transition between the states of some finite state machine each application of Fold is like "rotating the crank" of this finite state machine, and we will rotate it until there are elements left in the second argument list (this analogy belongs to David Wagner). The number of times that the function will be nested, is equal to the length of the second argument list. The function FoldList is related to Fold in the same way as NestList to Nest - it gives all intermediate results of "rotating the crank".

One very frequent use of Fold is recursion removal. This we will cover later in part II, chapter in recursion and iteration.

5.5.2 More examples:

5.5.2.1 Example: partial sums

```
FoldList[Plus, 0, {a, b, c, d, e, f}]
```

```
{0, a, a + b, a + b + c, a + b + c + d, a + b + c + d + e, a + b + c + d + e + f}
```

Here, as compared to Fold, we get all the intermediate partial sums for free. At the same time, FoldList is almost as efficient as Plus@@ for the final sum:

```
FoldList[Plus, 0, Range[100]]
{0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120,
 136, 153, 171, 190, 210, 231, 253, 276, 300, 325, 351, 378,
 406, 435, 465, 496, 528, 561, 595, 630, 666, 703, 741, 780,
 820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275,
 1326, 1378, 1431, 1485, 1540, 1596, 1653, 1711, 1770, 1830,
 1891, 1953, 2016, 2080, 2145, 2211, 2278, 2346, 2415, 2485,
 2556, 2628, 2701, 2775, 2850, 2926, 3003, 3081, 3160, 3240,
 3321, 3403, 3486, 3570, 3655, 3741, 3828, 3916, 4005, 4095,
 4186, 4278, 4371, 4465, 4560, 4656, 4753, 4851, 4950, 5050}
FoldList[Plus, 0, Range[100]] // myTiming
0.0000431
Plus @@ Range[100] // myTiming
0.000037
```

The Total command is much faster on the small lists, however:

```
Total[Range[100]] // myTiming
0.0000121
```

- 5.5.2.2 Example: position intervals for list splitting

Imagine that we are given a list of elements, and a list of partial lengths, which add to the length of the element list. The problem is to generate the list of position intervals, corresponding to splitting of the element list into sublists with the lengths given by the second list. It is not hard to convince oneself that the solution is given by:

```
Clear[splitIntervals];
splitIntervals[x_List, {lengths__Integer ? NonNegative}] /;
   Total[{lengths}] == Length[x] :=
  Transpose[{Most[#], Rest[#] - 1} &[FoldList[Plus, 1, {lengths}]]];
```

The main idea is to generate the list of start and end points of the intervals from the partial sums of lengths, and then transpose them to get the intervals. For example :

```
splitIntervals[Range[10], {2, 3, 5}]
{{1, 2}, {3, 5}, {6, 10}}
```

- 5.5.2.3 Application: splitting the list into sublists of specified lengths (generalized Take operation)

We can put the just developed function into a direct use to create a quite useful one: given a list and the list of partial lengths, split the original list into sublists of these lengths. Here is the code :

```
Clear[splitList];
splitList[x_List, {lengths__Integer ? NonNegative}] /;
   Total[{lengths}] == Length[x] :=
  Map[Take[x, #] &, splitIntervals[x, {lengths}]];
```

Check :

```
splitList[Range[20], {2, 1, 5, 6, 2, 4}]
{{1, 2}, {3}, {4, 5, 6, 7, 8},
 {9, 10, 11, 12, 13, 14}, {15, 16}, {17, 18, 19, 20}}
```

The final comment here is that this is an example of redundant error - checking as discussed at the end of section 5.2.7.3.2. To eliminate it in this case, it is best to embed the body of the < splitIntervals > function inside the < splitList > function, since it is really short (if we would have to write several functions using < splitIntervals >, we would perhaps be better off putting everything into a package and making < splitIntervals > a private (internal) function - then we can remove type checks):

```
Clear[splitListAlt];
splitListAlt[x_List, {lengths__Integer ? NonNegative}] /;
  Total[{lengths}] == Length[x] := Map[Take[x, #] &,
  Transpose[{Most[#], Rest[#] - 1} &[FoldList[Plus, 1, {lengths}]]]]
```

To see how much overhead the redundant type - checks induce in this case, we can perform a power test :

```
splitList[Range[Total[Range[100]]], Range[100]] // myTiming
0.0014
splitListAlt[Range[Total[Range[100]]], Range[100]] // myTiming
0.00121
```

We see that both functions are quite fast (this is one of the fastest implementations of this function in *Mathematica* that I know of), and the difference is of the order of 10 %. The difference would be more considerable if the patterns used in error - checking were more semantic (here they are mostly syntactic).

5.5.2.4 Example: imitating a factorial function

This is a (admittedly, rather inefficient) simulation of the factorial function with Fold.

```
Clear[ourFactorial];
ourFactorial[n_Integer ? NonNegative] := Fold[Times, 1, Range[n]];
```

For instance,

```
ourFactorial /@ Range[10]
{1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}
```

If, however, we have to generate a list of all consecutive factorials, then all we need to do is to change Fold to FoldList, and in this case this solution will be among the most efficient ones in *Mathematica*.

5.5.2.5 Example: imitating FromDigits

Here we will use Fold to imitate the <FromDigits> command. We are given a list of digits of a number, and the integer base. We have to reconstruct the number. If the base is not given, we should assume that it is 10. For example:

```
FromDigits[{2, 4, 5, 2, 3}]
24 523
FromDigits[{1, 0, 0, 1}, 2]
9
```

Here is the code using Fold:

```
Fold[#1 * 10 + #2 &, 0, {2, 4, 5, 2, 3}]
24 523
```

Here, we feed the digits one by one to a function which multiplies the previous result by 10 and adds a new digit at each step. It is clear how the code for the function will look like:

```
Clear[ourFromDigits];

ourFromDigits[digits_List] := Fold[#1 * 10 + #2 &, 0, digits];
ourFromDigits[digits_List, base_Integer ? Positive] :=
  Fold[#1 * base + #2 &, 0, digits];
```

Check:

```
ourFromDigits[{2, 4, 5, 2, 3}]
24 523
ourFromDigits[{1, 0, 0, 1}, 2]
9
```

We can express this as a single function by using the default values mechanism (optional patterns):

```
Clear[ourFromDigitsAlt];
ourFromDigitsAlt[digits_List, base_Integer : 10] :=
  Fold[#1 * base + #2 &, 0, digits];
```

Check now :

```
ourFromDigitsAlt[{2, 4, 5, 2, 3}]
24 523
ourFromDigitsAlt[{1, 0, 0, 1}, 2]
9
```

An interested reader may perform the efficiency analysis to see how close we get to the built-in FromDigits, for large numbers.

5.5.2.6 Example: powers of a differential operator

Here we consider the following problem: given a differential operator <oper>, construct an operator which will apply <oper> raised to some integer power, to a function <f>. However, we want to keep everything in an operator form: no explicit variables should be involved. The operator <oper> has to take a function <f>, and produce a pure function which corresponds to <oper>[f]. The new operator <oper-Power> should take an operator <oper>, a function <f> , and an integer power <n>, and produce a pure function <oper>^n[f]. Once any specific argument (say, <x> or <y> etc) is supplied, this has to evaluate to a function of this argument.

To be specific, consider the following operator <oper>:

```
HoldForm[x * f + D[f, x]] // TraditionalForm
```

$$f\,x+\frac{\partial f}{\partial x}$$

This is the code for it:

```
Clear[oper];
oper[expr_] := (# * expr + D[expr, #]) &;
```

For instance, this is the result of the single application of an operator:

```
oper[Sin[x]][x]
```

```
Cos[x] + x Sin[x]
```

What is important is that the action of an operator produces a function:

```
oper[Sin[y]][y]
```

```
Cos[y] + y Sin[y]
```

Note that <oper> defined in this way is vulnerable to misuse:

```
oper[Sin][x] // FullForm
```

```
Times[Sin, x]
```

```
oper[Sin[x]][y]
```

```
y Sin[x]
```

In both cases above the output is not what we would like. We should then consider it an auxiliary function, an input to <operPower>. Only the <operPower> should be used, and in particular it will give back the correct form of the operator <oper> if the power <n> =1.

Here is the code for operPower:

```
Clear[operPower];
operPower[oper_, f_, n_Integer ? NonNegative] := Module[{x},
    Fold[oper[#1][#2] &, f[x], Table[x, {n}]] /. x → #] &;
```

The code is somewhat tricky. The idea is that we use a local variable <x> and create a list like {x,x,...} of the length n. At each step, we supply it to <oper[#1][#2]&> as a second argument. But let us realize, that the result of the operation of <oper> at each step is a function, this is why we need to supply <x> to it as an argument to get an expression, and the syntax contains two pairs of square brackets one after another. After Fold has finished, we have an expression depending only on x. We then use a substitution rule to convert the resulting expression depending on local <x> to a pure function, which is then the final result.

Check, for instance:

```
operPower[oper, Sin, 0][x]
Sin[x]
operPower[oper, Sin, 1][x]
Cos[x] + x Sin[x]
operPower[oper, Sin, 2][x]

{Sin[x], Cos[x] + x Sin[x], x Cos[x] + x (Cos[x] + x Sin[x])}
```

The last case we may check also by hand:

```
x * (x * Sin[x] + D[Sin[x], x]) + D[x * Sin[x] + D[Sin[x], x], x]
x Cos[x] + x (Cos[x] + x Sin[x])
```

We can also use pure functions with <operPower>:

```
operPower[oper, #^2 &, 3][x]
```

$$8 x + x^3 + x \left(2 + 3 x^2\right) + x \left(2 + 3 x^2 + x \left(2 x + x^3\right)\right)$$

The advantage of this seemingly perverse solution is that we may separate functions from the arguments, and effectively carry operations on functions (internally, the arguments are used, but the user of this command does not need to know it).

Another comment: had we used FoldList instead of Fold, and we would get all the intermediate powers of our differential operator for free.

```
Clear[oper, operpower];
```

- 5.5.2.7 Example: autocorrelated random walks

Consider a following problem: we have a random walker, who can make unit steps to the left and to the right. Let the probability of step in each direction depend on the direction of the previous step. We may introduce a parameter < p > which tells how (anti) correlated or uncorrelated is our random walk. If < p > is close or equal to 0.5, the walk is almost or totally uncorrelated since the probability of a step to the right or to the left is the same and equal to 0.5. If < p > is close or equal to 1, the walk is almost or totally correlated since the next step will almost certainly be in a direction of the previous step. If < p > is close to or equal to 0, then the walk is almost or totally anti - correlated since the next step will almost certainly be in the direction opposite to the previous step.

```
Clear[randomStep];
randomStep[p_Real, previous : 1 | -1] /; 0 ≤ p ≤ 1 :=
 If[previous == 1, #, -#] &@
  Switch[p - Random[], _ ? Negative, -1, _ ? NonNegative, 1]
```

Correlated situation

```
steps = NestList[randomStep[0.7, #] &, 1, 40]
```

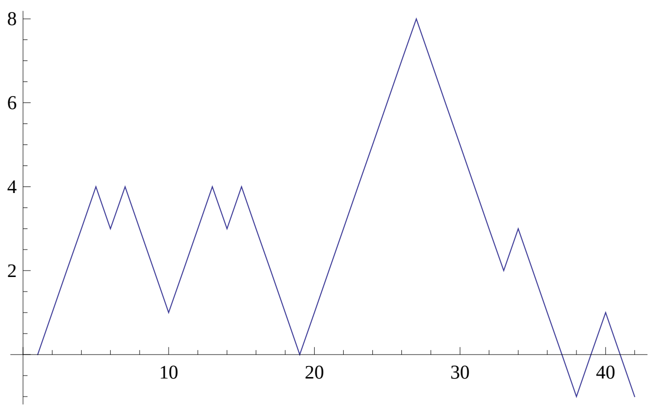
```
{1, 1, 1, 1, -1, 1, -1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1,
 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, 1, -1, -1}
```

We need FoldList to build a trajectory from the steps :

```
trajectory = FoldList[Plus, 0, steps]
```

```
{0, 1, 2, 3, 4, 3, 4, 3, 2, 1, 2, 3, 4, 3, 4, 3, 2, 1, 0, 1, 2,
 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 3, 2, 1, 0, -1, 0, 1, 0, -1}
```

```
ListPlot[trajectory, PlotJoined → True]
```



Anti - correlated situation

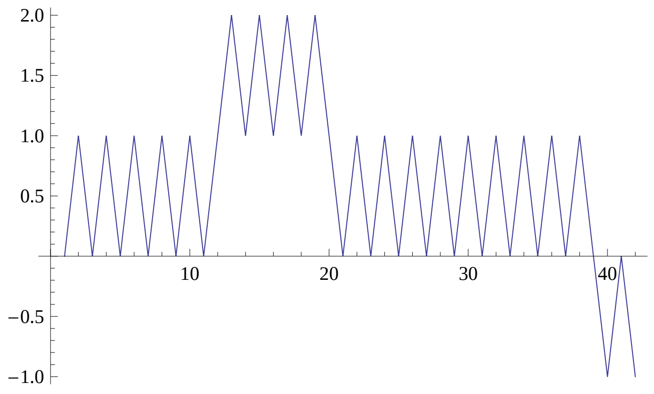
```
steps = NestList[randomStep[0.1, #] &, 1, 40]
```

```
{1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, -1, 1,
 -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, -1, 1, -1}
```

```
trajectory = FoldList[Plus, 0, steps]
```

```
{0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 2, 1, 2, 1, 2, 1, 2, 1, 0,
 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1}
```

```
ListPlot[trajectory, PlotJoined → True]
```



■ 5.5.2.8 Example: linked lists and the fast accumulation of results

For many applications, one needs to be able to build up a list of some intermediate results obtained in some computation. The easiest way to set up such a list is to use Append or Prepend (or perhaps, AppendTo or PrependTo). However, for large lists this method is quite inefficient. The reason is that lists in *Mathematica* are implemented as arrays, and thus every time we add an element, the entire list is copied.

We can use FoldList to illustrate the creation of a list in such manner :

```
testlist  = Table[Random[Integer, {1, 15}], {10}]
{5, 11, 10, 14, 5, 12, 15, 10, 6, 12}
FoldList[Append, {}, testlist]
{{}, {5}, {5, 11}, {5, 11, 10}, {5, 11, 10, 14},
 {5, 11, 10, 14, 5}, {5, 11, 10, 14, 5, 12},
 {5, 11, 10, 14, 5, 12, 15}, {5, 11, 10, 14, 5, 12, 15, 10},
 {5, 11, 10, 14, 5, 12, 15, 10, 6}, {5, 11, 10, 14, 5, 12, 15, 10, 6, 12}}
```

Now, let us do some performance tests :

```
Fold[Append, {}, Range[100]]; // Timing
{0., Null}
Fold[Append, {}, Range[500]]; // Timing
{0.01, Null}
Fold[Append, {}, Range[2000]]; // Timing
{0.06, Null}
Fold[Append, {}, Range[5000]]; // Timing
{0.41, Null}
Fold[Append, {}, Range[20 000]]; // Timing
{5.999, Null}
```

We see that the time used by this operation is quadratic in the size of the list. We of course would like a linear time. One way to achieve this which is available starting with the *Mathematica* version 5.0 is to use the Reap-Sow technique (to be described in Part II). Another (perhaps, slightly less efficient) way to get a linear time is to use linked lists. We will follow the discussion in the book of David Wagner [7].

```
Clear[a, b, c, d];
```

A linked list in *Mathematica* is a structure of the type

```
{a, {b, {c, {d, {}}}}}
```

The advantage of this representation is that on every level, we have a list containing just 2 elements, which is easy to copy. It will not work in this way for elements that are lists themselves, but then one can replace a list by an arbitrary head <h>.

```
Clear[h];
h[a, h[b, h[c, h[d, h[]]]]]
```

To avoid a possible conflict with some < h > already defined, we can use Module[{h}, ...] to make it local.

Using Fold is the most natural way to create such structures :

```
ll1 = Fold[{#2, #1} &, {}, Reverse[{a, b, c, d}]]

{a, {b, {c, {d, {}}}}}
ll2 = Fold[h[#2, #1] &, h[], Reverse[{a, b, c, d}]]
h[a, h[b, h[c, h[d, h[]]]]]
```

Converting them back to a normal list is just as easy with Flatten :

```
Flatten[ll1]
{a, b, c, d}
Flatten[ll2, Infinity, h]
h[a, b, c, d]
```

Notice that in the second case we used the fact that Flatten takes as an optional third argument the head which has to be Flatten - ed, and then Flatten - s only subexpressions with this head. In any case, this is another linear-time operation.

We can now write a function:

```
Clear[toLinkedList];
toLinkedList[x_List] := Fold[{#2, #1} &, {}, Reverse[x]];
```

Let us do some performance tests:

```
toLinkedList[Range[2000]]; // Timing
{0., Null}
toLinkedList[Range[5000]]; // Timing
{0.02, Null}
```

```
(ll3 = toLinkedList[Range[20 000]]); // Timing
{0.051, Null}
```

We see that the time is roughly linear in the list size, and for example, for a list of 20000 we get already a speed - up of the order of 100 times! Flattening is even faster:

```
Flatten[ll3]; // Timing
{1.31145×10^-14, Null}
```

Here we assumed that the list of results is accumulated immediately, just to separate this topic from the other problem - specific part of a program. If the list is accumulated not immediately but some other operations are performed in between (which is what usually happens), one just has to use the idiom **list = {newelement, list}**, to achieve the same result.

- 5.5.2.9 Example: joining linked lists

Continuing with linked lists, consider another problem : how to efficiently join many linked lists into a single one. For example :

```
ll1 = toLinkedList[Range[5]]
{1, {2, {3, {4, {5, {}}}}}}
ll2 = toLinkedList[Range[6, 10]]
{6, {7, {8, {9, {10, {}}}}}}
ll3 = toLinkedList[Range[11, 15]]
{11, {12, {13, {14, {15, {}}}}}}
```

We will now give a solution which works on lists of atoms. The key observation is that the empty list inside a linked list is always at level {-2}, and in the case of atomic lists, nothing else is on that level :

```
Level[ll2, {-2}]
{{}}
```

This means that if we want to join linked lists < x > and < y >, we just have to Map the entire list < y > on a level {-2} of the list < x > :

```
Map[ll2 &, ll1, {-2}]
{1, {2, {3, {4, {5, {6, {7, {8, {9, {10, {}}}}}}}}}}}
```

If we want to join many lists, we can do it pairwise, and use Fold, to join the previously joined list with a next list at every step. So, here is a final function :

```
Clear[lljoin];
lljoin[x_List, y_List] := Map[y &, x, {-2}];
lljoin[x__List] := Fold[lljoin, {}, {x}];
```

Check :

```
lljoin[ll1, ll2, ll3]
{1, {2, {3, {4,
    {5, {6, {7, {8, {9, {10, {11, {12, {13, {14, {15, {}}}}}}}}}}}}}}}}
```

5.5.3 Restriction of Fold-ed function to two arguments is spurious

Since the function to be Fold-ed can accept as a second (supplied externally by Fold) argument any expression, in particular a list, this means that we can actually supply as many arguments as we want. To illustrate this, consider the following example:

5.5.3.1 Example: random changes in the list

Here we take a list of 10 random integers in the range {1,100}. We then change the element at random position in this list by one of the symbols {a,b,c,d,e} (randomly chosen).

```
Clear[a, b, c, d, e];
```

Here is our list:

```
testlist = Table[Random[Integer, {1, 100}], {10}]
{60, 14, 18, 99, 24, 25, 69, 80, 4, 41}
```

```
changeList = Table[{{a, b, c, d, e}[[Random[Integer, {1, 5}]]] ,
   Random[Integer, {1, 10}]}, {10}]
{{c, 2}, {b, 5}, {c, 4}, {c, 7},
 {c, 2}, {b, 5}, {d, 2}, {c, 8}, {c, 6}, {a, 1}}
```

We will use a combination of ReplacePart and Fold:

```
FoldList[ReplacePart[#1, Sequence @@ #2] &, testlist, changeList]
{{60, 14, 18, 99, 24, 25, 69, 80, 4, 41},
 {60, c, 18, 99, 24, 25, 69, 80, 4, 41},
 {60, c, 18, 99, b, 25, 69, 80, 4, 41},
 {60, c, 18, c, b, 25, 69, 80, 4, 41}, {60, c, 18, c, b, 25, c, 80, 4, 41},
 {60, c, 18, c, b, 25, c, 80, 4, 41}, {60, c, 18, c, b, 25, c, 80, 4, 41},
 {60, d, 18, c, b, 25, c, 80, 4, 41}, {60, d, 18, c, b, 25, c, c, 4, 41},
 {60, d, 18, c, b, c, c, c, 4, 41}, {a, d, 18, c, b, c, c, c, 4, 41}}
```

We see that ReplacePart accepts 3 variables, not 2. What we did was to package the two variables {symbol·ToInsert,positionToReplace} into a list (single unit), and these lists were supplied to ReplacePart at every step. Plus, we used Apply[Sequence,...] idiom, which means - when package arrived, we can strip the wrapper to get the goodies.

```
Clear[testlist, changeList];
```

5.5.3.2 Example: running standard deviation for an increasing or running list of data

Consider the following problem: given a data sample of N points, for which the mean < mean > and the standard deviation < sigma > have been computed, we add one more data point. We would like to compute the mean and the standard deviation of the modified sample using only the old values of < mean > and < sigma >, the number of points < N > and the value of the new added data point. Obviously, if we do this, we save a lot since we will not need to use all our points and recompute these quantities from scratch. It is not difficult to show that this can be done according to the following formulas:

$$\bar{x}_{NEW} = \frac{N}{N+1}\bar{x}_{OLD} + \frac{1}{N+1}\text{new}$$

$$\bar{\sigma}_{NEW} = \sqrt{\frac{1}{N+1}\left(N\,\bar{\sigma}_{OLD}^{2} + \frac{N}{N+1}\left(\bar{x}_{OLD} - \text{new}\right)^{2}\right)}$$

Here are the implementations of the formula for the mean:

```
Clear[averIter];
averIter[prevaver_, n_, newel_] :=
  n / (n + 1) * prevaver + 1 / (n + 1) newel;
```

To check it, the code below computes the average of increasing list of natural numbers from {1} to {1, 2, 3, 4, 5} :

```
Module[{n = 0}, FoldList[averIter[#1, n ++, #2] &, 0, Range[5]]]
```

$$\left\{0, 1, \frac{3}{2}, 2, \frac{5}{2}, 3\right\}$$

Below is the function for a standard deviation. Notice that there are many ways of implementing it, but we deliberately made it such that it accepts three arguments: the previous mean and previous sigma combined together in a list, the previous number of elements and the new element, and returns a list of new mean and new sigma.

```
Clear[sigmaIter];
sigmaIter[{prevaver_, prevsigma_}, nold_Integer, newel_] :=
  Module[{newaver, newsigma},
   newaver = averIter[prevaver, nold, newel];
   newsigma = Sqrt[1 / (nold + 1) * ((nold * prevsigma^2)
        + nold / (nold + 1) (prevaver - newel ) ^2)];
   {newaver, newsigma}];
```

This is how we would use FoldList to generate pairs of {mean, sigma} for an increasing list of values (we use the same natural numbers as a test example) :

```
Module[{n = 0},
 Rest[FoldList[sigmaIter[#1, n ++, #2] &, {0, 0}, Range[5]]]]
```

$$\left\{\{1, 0\}, \left\{\frac{3}{2}, \frac{1}{2}\right\}, \left\{2, \sqrt{\frac{2}{3}}\right\}, \left\{\frac{5}{2}, \frac{\sqrt{5}}{2}\right\}, \left\{3, \sqrt{2}\right\}\right\}$$

Notice the use of n++ construct. This is a type of procedural code embedding which we already met discussing the Map function (see section 5.2.2.9). It is easy to check that this gives the same result as if we would use standard formulas:

```
Clear[meanSt, sigmaSt];
meanSt[x_List] := Total[x] / Length[x];
sigmaSt[x_List] := Sqrt[Total[(x - meanSt[x]) ^2] / Length[x]]
```

Note that in the part for <sigma> I used the vectorized nature (Listability) of Subtract and Power functions. Check:

```
Map[{meanSt[#], sigmaSt[#]} &, Range /@ Range[5]]
```

$$\left\{\{1, 0\}, \left\{\frac{3}{2}, \frac{1}{2}\right\}, \left\{2, \sqrt{\frac{2}{3}}\right\}, \left\{\frac{5}{2}, \frac{\sqrt{5}}{2}\right\}, \left\{3, \sqrt{2}\right\}\right\}$$

We can now package this into a function:

```
Clear[meanAndSigmaAccumulative];
meanAndSigmaAccumulative[x_List] := Module[{n = 0},
   Rest[FoldList[sigmaIter[#1, n ++, #2] &, {0, 0}, x]]];
```

We can see how much we win on large lists:

```
(ms1 = Map[{meanSt[#], sigmaSt[#]} &, Range /@ Range[500]]); //
 myTiming

0.461

(ms2 = meanAndSigmaAccumulative[Range[500]]); // myTiming

0.12

ms1 == ms2

True
```

The complexities are different so the difference will be larger for larger lists. Also, it is not as dramatic as in other languages because the second (general) implementation is helped by the fact that operations such as list subtraction or totalizing are Listable and highly optimized in *Mathematica*, whereas the one with FoldList necessarily breaks the process into pieces and thus looses this advantage.

Another situation where this may be useful is when, while one point is added to the sample, some other point is removed from it. This case can be treated in the same way (it is easy to derive the corresponding formulas), and will correspond to what we can really call the "running" standard deviation. The implementation of it is left as an exercise to the reader.

■ 5.5.4 Case study: Gram - Schmidt orthogonalization

■ 5.5.4.1 A crash course on linear algebra and Gram-Schmidt procedure

We are given a number of vectors of dimension <n> (there can be at most <n> linearly independent). Vectors will be represented as lists of length <n>. The dot product on these vectors is defined as a sum of products of individual components (we assume flat Euclidean metric). For example:

```
vec1 = {a1, b1, c1}
vec2 = {a2, b2, c2}
```

```
{a1, b1, c1}
{a2, b2, c2}
```

Then, the dot product:

```
Dot[vec1, vec2]
```

```
a1 a2 + b1 b2 + c1 c2
```

where we used a built-in function Dot. Equivalently, we may write (dot is a shorthand notation for Dot)

```
vec1.vec2
```

```
a1 a2 + b1 b2 + c1 c2
```

A norm of the vector is defined as a square root of the dot product of vector with itself. For instance, the norm of <vec1> will be

```
Sqrt[vec1.vec1]
```

$$\sqrt{a1^2 + b1^2 + c1^2}$$

Geometrically the norm of the vector is its length. The above result may be considered as a generalization of a Pythagorean theorem to n-dimensional space (n=3 in this case).
The vector is normalized when its norm is 1. Any vector can be normalized by dividing the vector by its norm. For instance,

```
vec1 / Sqrt[vec1.vec1]
```

$$\left\{\frac{a1}{\sqrt{a1^2 + b1^2 + c1^2}}, \frac{b1}{\sqrt{a1^2 + b1^2 + c1^2}}, \frac{c1}{\sqrt{a1^2 + b1^2 + c1^2}}\right\}$$

is normalized.

A linear combination of some number of vectors is a sum of these vectors multiplied by some coefficients - it is another vector. For instance, vector <vec3>

```
vec3 = α * vec1 + β * vec2
```

```
{a1 α + a2 β, b1 α + b2 β, c1 α + c2 β}
```

is a linear combination of <vec1> and <vec2> (α and β are some parameters). One may consider linear combinations of any number of vectors.

A set of vectors is said to be linearly independent if none of them can be represented as a linear combination of the others with some coefficients non-zero.

Any linearly-independent set of vectors defines a basis in a linear space, which is said to be spanned on these vectors. This means that any vector in this space can be written as a linear combination of the basis vectors, and vice versa, any vector which is a linear combination of these, belongs to this space. The dimension of this space is equal to a number of the vectors in the basis. To form a basis in the linear space of dimension <n>, we then need <n> vectors.

Two vectors are orthogonal when their dot product vanishes. The problem of orthogonalization of a set vectors consists of finding linear combinations of these vectors which are mutually orthogonal. Most of the time, one is interested in finding an orthonormal basis in a given linear space. In other words, given a number of (not necessarily orthogonal) linearly-independent vectors, we would like to find linear combinations of these vectors which will be mutually orthogonal.

The Gram-Schmidt orthogonalization procedure consists of the following steps:

1. Pick any vector from the initial set, call it v1.
2. Pick some other vector, call it v2.
3. Construct a new vector as v2New = v2 - (v2.v1)/(v1.v1) *v1; this new vector will be orthogonal to v1, as is easy to verify.
4. Pick another vector v3, and construct a new one as
 v3New = v3 - (v3.v1)/(v1.v1) *v1 - (v3.v2New)/(v2New.v2New)*v2New;
 this one will be orthogonal to both v1 and v2New (which is also easy to verify, remembering that v2New is orthogonal to v1).
5. The procedure goes on, until all the resulting vectors are mutually orthogonal.

This description may suggest that depending on the sequence in which we orthogonalize the vectors, we may get different sets of final orthogonal basis vectors. Although this is not obvious, all these sets will be equivalent (after the vectors are normalized) up to relabeling which vector is the first, second etc.

We will now implement a one step of this procedure. Assume that we have some number <m> of orthogonal vectors of length <n> already stored in the m x n matrix <vecmat>, and a new vector <vec> which we want to make orthogonal to those in the <vecmat>.

5.5.4.2 Implementing a single step of the Gram-Schmidt procedure

Here is the code for a single step of the Gram-Schmidt procedure:

```
Clear[oneStepOrtogonalize];
oneStepOrtogonalize[vec_List, {}] := vec;
oneStepOrtogonalize[vec_List, vecmat_ ? MatrixQ] :=
  Fold[(#1 - (vec.#2) / (#2.#2) * #2) &, vec, vecmat];
```

The first rule is added to include the case of the first vector - then it has to be simply returned back.

As a simple example, consider the matrix of vectors being vecmat = {{1,0,0},{0,0,1}}, and the vector to be made orthogonal to these, vec = {1,1,2}.

```
oneStepOrtogonalize[{1, 1, 2}, {{1, 0, 0}, {0, 0, 1}}]
{0, 1, 0}
```

The result is as we would expect - the missing basis vector. The way the code works is that the vector matrix is considered a list of second arguments to the function being Fold-ed . So, these second arguments will be vectors in this matrix. At each step, the initial vector gets transformed to be orthogonal to the vector supplied by Fold from the vector matrix at that step. When Fold is finished, the resulting vector is orthogonal to all of the vectors in the matrix.

- 5.5.4.3 Orthogonalization - the complete solution

Now that we worked out a single step, we have to get a complete solution, which will orthogonalize a given number of vectors.

Say, our initial vectors are

```
startVectors = {{1, 2, 3}, {5, 2, 7}, {3, 5, 1}}
{{1, 2, 3}, {5, 2, 7}, {3, 5, 1}}
```

The solution is immediate with another Fold:

```
result =
 Fold[Append[#1, oneStepOrtogonalize[#2, #1]] &, {}, startVectors]
```

$$\left\{\{1, 2, 3\}, \left\{\frac{20}{7}, -\frac{16}{7}, \frac{4}{7}\right\}, \left\{\frac{7}{3}, \frac{7}{3}, -\frac{7}{3}\right\}\right\}$$

What happens here is that a newly orthogonalized vector is appended to the current (initially empty) vector matrix. The new vector matrix is then used for orthogonalization of the next vector, etc. We can check that all the resulting vectors are mutually orthogonal:

```
Outer[Dot, result, result, 1]
```

$$\left\{\{14, 0, 0\}, \left\{0, \frac{96}{7}, 0\right\}, \left\{0, 0, \frac{49}{3}\right\}\right\}$$

Notice the use of Outer here - we generated at once all the dot products. Since we wanted the vectors inside the matrix <result> to be treated as single units (to be plugged into Dot), we used the more general form of Outer (See section 5.3.4.8). The numbers on the diagonal are the norms squared of the three vectors. All off-diagonal elements are zero as they should be for orthogonal vectors.

Now, we can package the code into a function:

```
Clear[GSOrthogonalize];
GSOrthogonalize[startvecs_?MatrixQ] :=
  Fold[Append[#1, oneStepOrtogonalize[#2, #1]] &, {}, startvecs];
```

And for completeness, we present once again the code for oneStepOrtogonalize:

```
Clear[oneStepOrtogonalize];
oneStepOrtogonalize[vec_List, {}] := vec;
oneStepOrtogonalize[vec_List, vecmat_ ? MatrixQ] :=
  Fold[(#1 - (vec.#2) / (#2.#2) * #2) &, vec, vecmat];
```

This is then our final implementation which solves the problem of Gram-Schmidt orthogonalization. It is concise, transparent and efficient (well, the purists will insist on eliminating Append. However, for large number of vectors, the cost of Append will be negligible w.r.t. the cost of <oneStepOrtogonalize>).

Note that the number of vectors to be orthogonalized may be smaller than the size of the vectors. But what happens if we try to orthogonalize more vectors than the size of the vector (or, dimension of the linear space)? For instance:

```
GSOrthogonalize[{{1, 2}, {5, 3}, {4, 7}}]
```

$$\left\{\{1, 2\}, \left\{\frac{14}{5}, -\frac{7}{5}\right\}, \{0, 0\}\right\}$$

We see that the last vector is zero. This is as it should be - there can not be more linearly independent vectors than the length of the vector. Notice that this case was an automatic consequence of our procedure and did not require any special treatment.

■ 5.5.4.4 Adding normalization

Now we would like to make the resulting vectors not only orthogonal, but orthonormal (orthonormalize - make them all be also normalized).

To do this, first define a function to normalize a vector:

```
Clear[normalize];
normalize[vec_List] := vec / Sqrt[Dot[vec, vec]];
```

And now define GSOrthoNormalize

```
Clear[GSOrthoNormalize];
GSOrthoNormalize[startvecs_ ? MatrixQ] :=
  Map[normalize, GSOrthogonalize[startvecs]];
```

Now check:

```
newresult = GSOrthoNormalize[startVectors]
```

$$\left\{\left\{\frac{1}{\sqrt{14}}, \sqrt{\frac{2}{7}}, \frac{3}{\sqrt{14}}\right\}, \left\{\frac{5}{\sqrt{42}}, -2\sqrt{\frac{2}{21}}, \frac{1}{\sqrt{42}}\right\}, \left\{\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right\}\right\}$$

What we did is just to Map <normalize> on the resulting vectors once they are obtained. It is easy to check that now all the vectors are orthonormal:

```
Outer[Dot, newresult, newresult, 1] // Simplify
{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

5.5.4.5 Application: generating random orthogonal matrices

As an application of the Gram-Schmidt method, we may consider a generation of random orthogonal matrices. The square matrix is called orthogonal if all the columns (rows) in it are normalized and mutually orthogonal. These matrices have important applications in various branches of science.

Basically, the procedure to generate a random orthogonal matrix is very simple: generate a plain random matrix, check that it has a non-zero determinant (if this is not so, it means that one of the columns - or rows - can be represented as a linear combination of the others. We will not consider such cases), and then simply apply our GSOrthoNormalize function to it. The result will be a random orthogonal matrix (this is of course an oversimplification. To unambiguously specify what we mean by a random matrix, we have to specify the probability distributions for all its elements. So, what we will generate here will be <some> random orthogonal matrix.
More specifically, the rows of this matrix are eigenvectors of the underlying one we start with, and for the latter one we will use the uniform [0,1] distribution for all matrix elements).

So, let us start with the example:

First generate a 3x3 random matrix

```
matrix = Table[Random[], {3}, {3}]
```

```
{{0.0647266, 0.478053, 0.613517},
 {0.803177, 0.140758, 0.584748}, {0.417063, 0.290149, 0.183038}}
```

Check the determinant

```
Det[matrix]
```

```
0.143951
```

Apply our function (if the determinant is zero with your run, please rerun the above until it is nonzero):

```
GSOrthoNormalize[matrix]
```

```
{{0.0829333, 0.612522, 0.786091},
 {0.946642, -0.294931, 0.129938}, {0.311432, 0.733371, -0.604299}}
```

Now we will package this into another function:

```
Clear[randomOrthogonal];
randomOrthogonal[n_Integer ? Positive] := GSOrthoNormalize[
   NestWhile[Array[Random[] &, {n, n}] &, {{0}}, Det[#] === 0 &]];
```

Here I used Array instead of Table, since it is slightly faster. But one could use Table as well. The use of NestWhile guarantees that the result of it will have a non-zero determinant. Check:

```
randomOrthogonal[3]
```

```
{{0.241594, 0.34434, 0.907228},
 {-0.0046949, -0.934497, 0.35594}, {-0.970366, 0.0902524, 0.224152}}
```

```
randomOrthogonal[5]
{{0.345974, 0.451951, 0.450486, 0.366318, 0.582164},
 {0.694379, -0.320171, 0.222314, 0.300302, -0.525094},
 {0.0645491, 0.798033, 0.0566894, -0.270187, -0.531753},
 {0.627364, 0.0304164, -0.521895, -0.485135, 0.312664},
 {-0.0196963, 0.235475, -0.687059, 0.683573, -0.0695739}}
```

■ 5.5.4.6 Performance analysis

This is how much time it takes to generate 50x50 random orthogonal matrix with this method:

```
randomOrthogonal[50]; // Timing
{0.08 Second, Null}
```

Assuming that the dot product is optimized and is roughly constant time for small vector sizes,we expect the complexity to be roughly quadratic with the matrix size for small matrices. This is because we have to ensure the mutual orthogonality of all vectors, and the number of vector pairs grows quadratically with the number of vectors. We can check our expectations:

```
randomOrthogonal[12] // myTiming
0.011
randomOrthogonal[25] // myTiming
0.0231
randomOrthogonal[50] // myTiming
0.12
randomOrthogonal[100] // myTiming
0.36
randomOrthogonal[400] // myTiming
5.898
```

We see that we actually get even a slightly better situation, due most likely to an optimized nature of Fold. One final comment: one can somewhat improve the performance by caching the norms of the vectors computed during the Gram-Schmidt procedure - we recompute them every time afresh. This can give a factor of 1.5~2 speed-up.

```
Clear[vec1, vec2, vec3, oneStepOrtogonalize, result, startVectors,
  GSOrthogonalize, newresult, normalize, GSOrthoNormalize];
```

■ 5.5.4.7 Appendix: the complete code for Gram-Schmidt orthogonalization

Here we just put the complete code (functions) into a single place:

```
Clear[oneStepOrtogonalize, GSOrthogonalize,
  normalize, GSOrthoNormalize, randomOrthogonal];
```

```
oneStepOrtogonalize[vec_List, {}] := vec;
oneStepOrtogonalize[vec_List, vecmat_ ? MatrixQ] :=
  Fold[(#1 - (vec.#2) / (#2.#2) * #2) &, vec, vecmat];

GSOrthogonalize[startvecs_ ? MatrixQ] :=
  Fold[Append[#1, oneStepOrtogonalize[#2, #1]] &, {}, startvecs];

normalize[vec_List] := vec / Sqrt[Dot[vec, vec]];

GSOrthoNormalize[startvecs_ ? MatrixQ] :=
  Map[normalize, GSOrthogonalize[startvecs]];

randomOrthogonal[n_Integer ? Positive] := GSOrthoNormalize[
   NestWhile[Array[Random[] &, {n, n}] &, {{0}}, Det[#] === 0 &]];
```

We see that the code is concise and transparent. This is an example of how the typical solution to a given problem looks in *Mathematica*.

5.5.5 Small case study: local maxima for a list

5.5.5.1 The problem

Sometimes, Fold (FoldList) allows for extremely concise and beautiful solutions. For example, here is the problem: for a given list of numbers, produce list of all its elements that are larger than any element before it, in this list (this is one of the favorite problems in many texts on *Mathematica* programming, so I decided to continue the tradition).

Here is our test list:

```
lst = Table[Random[Integer, {1, 40}], {25}]
{12, 13, 33, 20, 22, 12, 10, 15, 1, 17, 31,
 21, 38, 7, 36, 31, 35, 14, 1, 15, 20, 13, 25, 2, 1}
```

5.5.5.2 Procedural solution

It is fairly obvious how to write a procedural solution:

```
Clear[localMaxListProc];
localMaxListProc[x_List] :=
  Module[{i, temp, len = Length[x], reslist, solctr},
   For[reslist = Table[Null, {len}];
    temp = 0; i = solctr = 1, i ≤ len, i ++,
    If[temp < x[[i]], reslist[[solctr ++]] = temp = x[[i]]];];
   Drop[reslist, -(len -solctr +1)]];
```

Here, we preallocate the list of results, since we know that its length can be at most equal to the length of the initial list. As in cases before, this is done to avoid using Append and make a code more efficient. Check:

```
localMaxListProc[lst]
{12, 13, 33, 38}
```

■ 5.5.5.3 Functional solution with FoldList

This is essentially the same solution but expressed with FoldList:

```
Clear[localMaxList];
localMaxList[x_List] := Union[Rest[FoldList[Max, -Infinity, x]]];
```

Check:

```
localMaxList[lst]
{12, 13, 33, 38}
```

To see, what happens, we can dissect the function into pieces:

```
FoldList[Max, -Infinity, lst]
{-∞, 12, 13, 33, 33, 33, 33, 33, 33, 33, 33, 33,
 33, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38}
```

Here, at every iteration the result of the previous application of Max is compared with the next number from the list (we could have put Max[#1,#2]&), and the maximum of the two becomes the current result. Thus,the next result will only be different from the previous if we get a number larger than all encountered before. The value -∞ is used as a starting number, to guarantee that the starting number will be less than any in the list (which may also contain negative numbers).

At the next stage, we delete this number with the help of Rest:

```
Rest[FoldList[Max, -Infinity, lst]]
{12, 13, 33, 33, 33, 33, 33, 33, 33, 33, 33,
 33, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38}
```

Finally, we take the Union to eliminate duplicate elements. Union also sorts the results in an ascending order, but in our case this is just fine.

```
Union[Rest[FoldList[Max, -Infinity, lst]]]
{12, 13, 33, 38}
```

■ 5.5.5.4 Performance analysis

We can now compare the performance:

```
testlst = Table[Random[Integer, {1, 20 000}], {5000}];
localMaxListProc[testlst]; // Timing
{0.07, Null}
localMaxList[testlst]; // Timing
{0.02, Null}
```

We see that by using Fold, we gain a factor of 3-4 in this case (for this size of the test list), even though we took care to make the procedural realization efficient, and indeed naively it looks more efficient since it does not produce an entire list and then delete similar elements. However, what it does is to break an object (list) into pieces by using array indexing. And according to one of our rules of thumb, this practice should be avoided in *Mathematica*.

```
Clear[lst, testlst, localMaxList, localMaxListProc];
```

5.6 FixedPoint and FixedPointList

5.6.1 The syntax and functionality

These functions are very similar to Nest and NestList. Instead of asking, how many times the function should be nested, they nest the function until the result no longer changes. The format is

```
FixedPoint[f, expr]
```

In some cases, it is desirable to have a "safety net" - to stop nesting after certain maximal allowed number of iterations regardless of whether or not the result has stabilized. To account for these cases, there is an optional third argument <n> which gives a maximal number of iterations.

Since these functions are very similar to Nest/NestList except their termination condition, we will illustrate them by revisiting certain examples considered before.

5.6.2 Example: the Collatz problem revisited

Here is the already discussed (section 5.4.2.6) Collatz iteration definition:

```
Clear[c];
c[n_ ? OddQ] := 3 * n + 1;
c[n_ ? EvenQ] := n / 2;
```

Previously we solved the problem of generation of the Collatz sequence by using NestWhileList. The solution was:

```
Clear[colSequence];
colSequence[q_Integer] := NestWhileList[c, q, #1 ≠ 1 &];
```

Imagine for a moment that it is unavailable (like it was in earlier versions of *Mathematica*). Can we find a substitute? Here is the solution that uses FixedPointList:

```
Clear[colSequenceFP];
colSequenceFP[q_Integer] :=
  Drop[FixedPointList[If[# ≠ 1, c[#], 1] &, q], -1];
```

(here, FP stands for "FiexedPoint"). The idea is that after the result becomes one for the first time, it will remain one due to the way the nested function is written. Then, after the next iteration, the last two results will be both equal to 1 and to each other, and thus the process will stop. For instance

```
colSequence[233]
{233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890,
 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283,
 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367,
 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732,
 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46,
 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

```
colSequenceFP[233]
{233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890,
 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283,
 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367,
 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732,
 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46,
 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

The new solution is almost as efficient as the old one. The source of inefficiency here is that the condition that the result is one, is checked as every iteration. Note by the way that both the implementation with NestWhileList and FixedPointList (as well as these constructs themselves) are a kind of compromise, appreciating that procedural style is sometimes more appropriate than functional (this is, when at every time only very small and very "local" part of a given large structure (like list) is changed).

■ 5.6.3 How to reformulate a problem for FixedPoint

Basically, every problem where one has to generate some list by repeated nesting of a function <f>, and stop when certain condition <cond>is satisfied, can be reformulated as a problem for FixedPoint, in the following way:

```
FixedPoint[If[cond[#], #, f[#]] &, expr]
```

The logic here is the same as for the Collatz problem just considered.

■ 5.6.4 Example: deleting numbers from the list revisited

Here is a list containing in general zeros, positive and negative integers.

```
Clear[testlist];
testlist = Table[Random[Integer, {-5, 10}], {15}]
{7, 5, 4, 7, 6, -2, 2, -5, 0, 8, 6, -2, -4, 10, 10}
```

This will drop the first element in the list until it meets a first negative number - this is the solution we had before:

```
NestWhile[Drop[#, 1] &, testlist, NonNegative[First[#]] &]
{-2, 2, -5, 0, 8, 6, -2, -4, 10, 10}
```

The solution with FixedPoint:

```
FixedPoint[If[Negative[First[#]], #, Drop[#, 1]] &, testlist]
{-2, 2, -5, 0, 8, 6, -2, -4, 10, 10}
```

Note: the same **performance pitfall** which we discussed for the NestWhile - based solution of this problem (section 5.4.2.1), is also present here.

5.6.5 Example: approximating the square root of a number revisited

Here is the sequence that we implemented previously with Nest to get an approximation to a square root of a number:

$t_{n+1} = 1/2\,(t_n + A/t_n)$

This was our solution (we take a specific number 3, and a starting number 5., as before)

```
NestList[(# + 3 / #) / 2 &, 5., 5]
{5., 2.8, 1.93571, 1.74276, 1.73208, 1.73205}
```

Here our starting number was 5, and we used 5 iterations altogether. The list of intermediate results shows that this method converges quite fast. If we are interested in final result only, then we use Nest:

```
result = Nest[(# + 3 / #) / 2 &, 5., 5]
1.73205
```

Now, here is the solution with FixedPoint:

```
newresult = FixedPointList[(# + 3 / #) / 2 &, 5.]
{5., 2.8, 1.93571, 1.74276, 1.73208, 1.73205, 1.73205, 1.73205}
```

It is even simpler in this case - once the number stops changing , that's it! This is a type of problem where FixedPoint works most directly without modifications.

The rest of the discussion of that example can be trivially transferred to the version with FixedPoint.

5.6.6 FixedPoint dangers

When used without the "safety net", FixedPoint has an obvious danger of getting in a situation where it will never stop, because the results may happen to always be different. But even in the seemingly clear cases, this danger is not as far away as we may think. Let me use the same setting as in the previous example: we want to estimate the square root of 3, starting with a number 5. I will alter it very slightly - forget to put the dot after 5: use <5> instead of <5.>. And, let us use a safety net of 7 iterations maximum - just in case...

```
FixedPointList[(# + 3 / #) / 2 &, 5, 7]
```

```
{5, 14/5, 271/140, 132 241/75 880, 34 761 005 281/20 068 894 160, 2 416 609 026 561 556 205 761/1 395 229 871 759 180 117 920,
 11 679 998 372 406 206 859 888 818 654 055 765 884 968 321/6 743 450 204 443 114 328 549 416 278 518 687 526 674 240,
 272 844 723 958 823 282 612 658 628 263 013 873 237 021 400 356 408 160 879 ⋱
   437 661 848 300 809 304 604 412 291 841 /
  157 526 974 824 595 756 512 861 009 809 263 458 922 744 712 800 339 912 508 ⋱
   216 462 467 898 433 196 289 773 502 080}
```

This is an unpleasant surprise. The point is that the operations are now done with arbitrary precision integer arithmetic, and the two consecutive results will ***never*** be the same. This tells us once again that we have to ensure in every case, that "sameness" is defined the same (pun unintended) for us and for *Mathematica*. If in doubt - put a maximum on the total number of iterations (although for final version of any program, you will be better off using the final iteration number only if there are more compelling reasons to do so, related to the structure of the problem. This comment does not apply for programs which implement certain functionality for further use, such as packages - in such cases it is always a good idea to constrain the number of iterations by some maximum value).

■ 5.6.7 Small case study: merging overlapping intervals - Functional vs. Rule-based

■ 5.6.7.1 The problem and a rule-based solution

Consider the following problem: given a list of intervals, some of which may overlap (such that some intervals may be entirely contained in others), write a function which will merge together those intervals which overlap, and return a modified list of intervals. The input list of intervals must be ordered in the sense that non-overlapping intervals closer to the origin should be on the left of those further from it.

Here is an elegant and concise rule-based solution shown in "Mastering *Mathematica*" by John Gray.

```
Clear[mergeIntervals];
mergeIntervals[ints_List] := ints //.
   {e___, {a_, b_}, {c_, d_}, f___} /; c ≤ b ⧴ {e, {a, Max[b, d]}, f};
```

The code is almost self-explanatory. We saw already examples of this type of programming earlier - see the section 4.2.5. The use of //. (ReplaceRepeated) ensures that the rule will be applied until all overlapping intervals are merged. For instance:

```
mergeIntervals[{{1, 3}, {2, 4}, {5, 7}, {6, 8}, {7, 9}}]
```

```
{{1, 4}, {5, 9}}
```

■ 5.6.7.2 A functional solution

The above solution is definitely beautiful and clear. However, it is not the most efficient one, as we will see in a minute.

Let us see what we can do within the functional style. The first thing that comes to mind is to somehow group together overlapping intervals, keeping in mind that there can be chains of more than two intervals overlapping, like the last 3 intervals in the above example. The way to do it is to use a built-in <Split>

command. Split does the following: it groups together the "same" elements, and we can define what we mean by "same", by providing to Split a comparison function. The limitation of Split is however that it can only compare adjacent elements.

So, let us define two intervals to be "the same" (for Split), when the right end of the first interval is larger than the left end of the second one:

```
step1 =
 Split[{{1, 3}, {2, 4}, {5, 7}, {6, 8}, {7, 9}}, #1[[2]] ≥ #2[[1]] &]
{{{1, 3}, {2, 4}}, {{5, 7}, {6, 8}, {7, 9}}}
```

We see that the intervals have been correctly grouped. Now we need to extract the new borders corresponding to the merged intervals. To do this, I suggest that we Flatten each group first (which means we have to Map Flatten on our list):

```
step2 = Flatten /@ step1
{{1, 3, 2, 4}, {5, 7, 6, 8, 7, 9}}
```

Now we can Sort each Flattened sublist:

```
step3 = Sort /@ step2
{{1, 2, 3, 4}, {5, 6, 7, 7, 8, 9}}
```

It now remains to take the first and last element of each sublist -since they are sorted, these will be the new borders:

```
step4 = #[[{1, -1}]] & /@ step3
{{1, 4}, {5, 9}}
```

This is the result in this case. We can now put all the operations together:

```
Sort[Flatten[#]][[{1, -1}]] & /@
 Split[{{1, 3}, {2, 4}, {5, 7}, {6, 8}, {7, 9}}, #1[[2]] ≥ #2[[1]] &]
{{1, 4}, {5, 9}}
```

This seems to solve the problem, but there is one subtle point. Consider the intervals: {{5,9},{6,7},{8,10}}. Let us see what we get:

```
Sort[Flatten[#]][[{1, -1}]] & /@
 Split[{{5, 9}, {6, 7}, {8, 10}}, #1[[2]] ≥ #2[[1]] &]
{{5, 9}, {8, 10}}
```

The problem is, that the resulting two intervals overlap and should have been combined into a single one, but this did not happen. The reason is that at the time when Split was grouping elements, the intervals {6,7} and {8,10} did not overlap - the overlap is induced by a previous interval {5,9} which entirely contains {6,7}. But since Split is limited to compare only adjacent elements, there seems to be no way out. Our solution then will be the following: use the code above as a "single step" transformation, and apply it repeatedly to the list until the list stops changing. Here is the code for a single step merging:

```
Clear[mergeIntsOneStep];
mergeIntsOneStep[ints_List] :=
  Map[Part[Sort[Flatten[#]], {1, -1}] &,
   Split[ints, #1[[2]] ≥ #2[[1]] &]];
```

And this is all we need to add to solve the problem:

```
Clear[mergeInts];
mergeInts[ints_List] := FixedPoint[mergeIntsOneStep, ints];
```

Check now:

```
mergeInts[{{1, 3}, {2, 4}, {5, 7}, {6, 8}, {7, 9}}]
{{1, 4}, {5, 9}}
mergeInts[{{5, 9}, {6, 7}, {8, 10}}]
{{5, 10}}
```

- 5.6.7.3 Comparing performance

Let us now compare the efficiency of the rule-based and procedural solutions, on interval lists of different lengths. We will generate these lists randomly. The code to generate the lists is as follows:

```
Clear[randomIntervals];
randomIntervals[min_Integer, max_Integer,
   maxrange_Integer ? Positive, intnum_Integer ? Positive] :=
  Module[{x}, Sort[Table[{x = Random[Integer, {min, max}],
      x + Random[Integer, {1, maxrange}]},
     {intnum}], #1[[1]] < #2[[1]] &]];
```

For instance, this will be a list of 10 intervals with borders from 1 to 100 and maximal length of an interval 8:

```
testints = randomIntervals[1, 100, 8, 10]
{{3, 9}, {14, 16}, {17, 24}, {24, 28}, {27, 32},
 {59, 65}, {76, 81}, {86, 90}, {97, 101}, {100, 101}}
```

We now test the performance:

```
mergeIntervals[testints] // myTiming
0.000281
mergeInts[testints] // myTiming
0.00025
```

For this very small list, the performance is roughly the same. Let us increase the number of intervals:

```
(testints = randomIntervals[1, 1000, 15, 100]) // Short
{{10, 11}, ≪98≫, {990, 995}}
```

Now:

```
mergeIntervals[testints] // myTiming
0.017
mergeInts[testints] // myTiming
0.0025
```

Here we see that our functional version is already 5 times faster. Let us increase even more:

```
(testints = randomIntervals[1, 4000, 25, 500]) // Short
{{13, 17}, ≪498≫, {3997, 4012}}
mergeIntervals[testints] // myTiming
0.38
mergeInts[testints] // myTiming
0.00811
```

Here the difference is more than an order of magnitude. What happens is that these solutions have different computational complexities. The reason is that the functional solution is able to group more than two overlapping intervals together in a single run through the list, while the rule-based solution can only group two adjacent intervals at a time. Thus, it needs more runs through the list, and as the length of the list increases, we pay for it more and more. In other words, the inefficiency is because it repeatedly checks non-overlapping intervals for matching the overlap pattern. In terms of our rules of thumb, this is a case for this one: "avoid inefficient patterns".

■ 5.6.7.4 Concluding remarks

So, what should be our conclusion? Is the rule-based solution necessarily bad, should we avoid it always and at all costs? I don’t think so. It is certainly easier to write and it communicates the idea more clearly. What is important is to develop the skill to recognize inefficiency in the code, so that, if a given part of the code turns out to be time-critical, you will be able to quickly spot the "bottleneck" and rewrite the code more efficiently.

```
Clear[mergeIntervals, mergeIntsOneStep,
  mergeInts, testints, randomIntervals];
```

■ 5.6.8 Example: local (relative) maxima in a list revisited

Let us revisit a problem of finding a list of local maxima for a given list - that is, a list of all elements which are larger than any element before it in the input list. We have given a solution using Fold (the best we have) and a procedural solution, in the chapter on Fold. Now we will consider a solution based on FixedPoint. It will not be as efficient, but it represents an interesting way of thinking in my view.

So, here is our test list :

```
tlist = Table[Random[Integer, {1, 40}], {20}]
{11, 6, 28, 21, 10, 9, 7, 22, 6, 37, 20, 5, 27, 30, 7, 3, 19, 35, 30, 8}
```

Consider a following transformation of this list : we will Split it into sublists of decreasing (non - increasing) elements, and then only keep a maximum (the left - most element) of each sublist :

```
step1 = Split[tlist, GreaterEqual]
{{11, 6}, {28, 21, 10, 9, 7}, {22, 6},
 {37, 20, 5}, {27}, {30, 7, 3}, {19}, {35, 30, 8}}
step1 = Map[First, step1]
{11, 28, 22, 37, 27, 30, 19, 35}
```

Or, we can combine this as :

```
step1 = Map[First, Split[#, GreaterEqual]] &[tlist]
{11, 28, 22, 37, 27, 30, 19, 35}
```

Now, we can iterate a few times :

```
step2 = Map[First, Split[#, GreaterEqual]] &[step1]
{11, 28, 37, 30, 35}
step3 = Map[First, Split[#, GreaterEqual]] &[step2]
{11, 28, 37, 35}
```

It is clear that this is a task for FixedPoint :

```
FixedPoint[Map[First, Split[#, GreaterEqual]] &, tlist]
{11, 28, 37}
```

So, our function will look like :

```
Clear[localMaxList];
localMaxList[x_List] :=
  FixedPoint[Map[First, Split[#, GreaterEqual]] &, x];
```

Let me note once again that this solution is less efficient than those we considered before, since it needs many iterations. However, it represents an interesting way of thinking which may be efficient in some other situations.

5.7 Operators on functions

5.7.1 Through

5.7.1.1 Syntax and functionality

This function is quite useful at times, although generally used less than those we discussed already. Actually, my feeling is that it is used less than it deserves to be. The format is

```
Through[p[f1, f2, ..., fn][x]]
```

The result is

```
Clear[p, f, g, h, x];
Through[p[f, g, h][x]]
p[f[x], g[x], h[x]]
```

5.7.1.2 Initial examples

As I said, this can be quite useful. For instance:

```
Through[(Sin*Cos)[x]]
Cos[x] Sin[x]
```

Or, when we have a list of functions:

```
Through[{f1, f2, f3}[x]]


{f1[x], f2[x], f3[x]}
```

5.7.1.3 When it goes wrong

But be careful - this for example may be a surprise:

```
Through[(Sin / Cos)[x]]
```

$\frac{1}{\mathrm{Cos}}[x]\ \mathrm{Sin}[x]$

The last output is such because division has been internally rewritten as a multiplication by the inverse, as can be seen from the FullForm:

```
FullForm[Sin / Cos]
Times[Power[Cos, -1], Sin]
```

There is an alternative (for this particular case) which will do what we failed to accomplish with Through:

```
Map[If[Head[#] === Symbol, #[x], #] &, Sin / Cos, {-1}]
Tan[x]
```

The result has been simplified to Tan[x] immediately.

Let me now illustrate the use of Through in a less trivial setting, where it is really useful.

■ 5.7.1.4 Better example: picking list elements randomly with prescribed probabilities

Here is a problem: given a list of elements, like for instance:

```
Clear[a, b, c, d, e, f, pl, rl, action];
rl = {a, b, c, d, e, f};
```

And a list of probabilities of picking these objects (which have to add to 1), like for instance

```
pl = {0.1, 0.2, 0.1, 0.15, 0.25, 0.2};
```

Write a function which will randomly pick an object from the list according to these probabilities.

The idea of the solution will be the following: we will split the interval [0,1] of the real line according to these probabilities. Then, we will generate a random number in the range [0,1], assuming the uniform distribution of the numbers for the built-in pseudo-random generator. We will then analyze, in which interval the generated number ends up being, and will pick the corresponding element of the list.

Let us first define an auxiliary function called <ineq>, which will take as two arguments two ends of the interval and return a pure function that checks whether a number belongs to this interval (pay attention - it returns a *function*)

```
Clear[ineq];
ineq[x_, y_] := (x < # ≤ y &)
```

Check:

```
fn15 = ineq[1, 5]
1 < #1 ≤ 5 &
```

```
fn15 /@ {1, 3, 5}
{False, True, True}
```

Now, let us partition an interval [0,1] into several intervals, according to the probabilities. We will first create a list of partial sums with FoldList:

```
margins = FoldList[Plus, 0, pl]
{0, 0.1, 0.3, 0.4, 0.55, 0.8, 1.}
```

These are the margins of our intervals. Now, let us use Partition to create the intervals:

```
intervals = Partition[margins, 2, 1]
{{0, 0.1}, {0.1, 0.3}, {0.3, 0.4}, {0.4, 0.55}, {0.55, 0.8}, {0.8, 1.}}
```

We now Map-Apply the <ineq> function to this list (we need Apply since it takes a sequence of 2 arguments rather than a list)

```
funs = ineq @@@ intervals
{0 < #1 ≤ 0.1 &, 0.1 < #1 ≤ 0.3 &, 0.3 < #1 ≤ 0.4 &,
 0.4 < #1 ≤ 0.55 &, 0.55 < #1 ≤ 0.8 &, 0.8 < #1 ≤ 1. &}
```

What we have here is a list of pure functions which check whether the argument belongs to a particular interval. This is the time to generate a random number:

```
rnd = Random[]
0.979687
```

Now, we use Through:

```
results = Through[funs[rnd]]
{False, False, False, False, False, True}
```

We will now use Position to find the position of <True>

```
pos = Position[results, True]
{{6}}
```

Finally, we will use Extract to extract the corresponding element:

```
First[Extract[rl, pos]]
f
```

We will now package everything into a function which we will call <pickObject>

```
Clear[pickObject];
pickObject[objs_List, probs_List] /;
   And[Length[objs] == Length[probs], Plus @@ probs == 1] :=
  First[Extract[objs, Position[Through[(ineq @@@ Partition[
          FoldList[Plus, 0, probs], 2, 1])[Random[]]], True]]];
```

Let us check:

```
pickObject[rl, pl]
c
```

```
Table[pickObject[rl, pl], {10}]
{e, d, b, b, b, a, e, f, e, f}
```

Let us gather some statistics:

```
stat = Table[pickObject[rl, pl], {1000}];
```

These are the probabilities (to remind)

```
pl
{0.1, 0.2, 0.1, 0.15, 0.25, 0.2}
```

We now count how many times each of the elements was picked:

```
Count[stat, #] & /@ {a, b, c, d, e, f}
{102, 202, 96, 160, 252, 188}
```

We see that we are not too far from the prescribed probabilities.

```
Clear[ineq, pl, rl, margins, intervals,
  funs, rnd, fn15, results, pos, stat, pickObject];
```

■ 5.7.2 Operate

This is perhaps an even more exotic operation. Basically, Operate applies some function to the head of an expression, using it as data:

```
Clear[f, g, x, y];
Operate[f, g[x, y]]
```

```
f[g][x, y]
```

So, <f> plays a role of a meta-function defined on <g>. For example:

```
f[Sin] = Cos;
Operate[f, Sin[x]]
```

```
Cos[x]
```

This function may be useful for instance in the following situation: you have created several "containers" for elements and given them some names (heads). Then you write a function which will perform different operations on the objects depending on in which container they are. Then the function <f> in Operate may serve as a dispatcher which will tell which operation to perform depending on the container name. But this is a somewhat different style of programming, which will take more accent to data from functions.

```
Clear[f];
```

5.8 Summary

This chapter has been very important in many aspects. For one thing, we considered a large number of examples which illustrated many subtle points of *Mathematica* programming.

Another good thing is that we covered in details most of important built-in higher-order functions, which serve as building blocks of most *Mathematica* programs. We have now a toolbox ready to be used for larger or more complicated problems.

Yet another important thing is that starting with this chapter, we systematically emphasized efficiency. I tried to convey the style of programing where efficiency considerations are used from the very beginning and all the way through solving the problem, but on the other hand not to hide the essence of the problem by efficiency analysis.

Also, we went many times through typical stages of development process for *Mathematica* programs. It is amazingly easy to develop a program in *Mathematica*: we start with a very simple test case, each step is usually just one line, each step is easily tested, the final code is trivially combined from the steps of the sample solution.

But most importantly, all the considerations of this chapter taken together hopefully illustrated the functional style of programming as not just a number of clever tricks but as an entirely different way to think about programming. This is the style that will be used most frequently and heavily in the chapters that follow, so it is very important that this material is well-understood.

VI. Writing efficient programs: some techniques and applications

6.1 Introduction

This last chapter of this part serves to illustrate the relative efficiency of different programming styles in *Mathematica* on several non - trivial applications. This chapter is somewhat different from the previous ones in style - it is somewhat less pedagogical - I don't explain every line of code in such detail as before (I assume that the interested reader who made it that far will be able to understand the workings of the code using my rather brief explanations as hints). But it shows real problems, wins and trade-offs that one deals with in more serious *Mathematica* programming. Also, some of the techniques discussed here are rather general and may be used in many other situations. Finally, the problems I discuss may be of interest by themselves.

However, there is a lot more to performance tuning in *Mathematica* than what is discussed in this chapter. I will have more to say about it in other parts of the tutorial. Excellent treatment of performance-tuning techniques is given in the book of David Wagner.

6.2 Case study I: checking if a square matrix is diagonal

6.2.1 The problem

The formulation of the problem is extremely simple: given a square matrix of some size, return True if all the off - diagonal elements are zero and False otherwise.

6.2.2 The test matrices

Here we will introduce relatively large test matrices : a random matrix which is almost certainly not diagonal, and an identity matrix of the same size, which is of course diagonal. All our implementations of DiagonalQ will be tested on these matrices.

```
testmatr = Array[Random[] &, {400, 400}];
testiden = IdentityMatrix[400];
```

6.2.3 Procedural implementation

The procedural implementation is straightforward

```
Clear[diagonalQProc];
diagonalQProc[m_ ? MatrixQ] /; Equal @@ Dimensions[m] :=
  Module[{len = Length[m], i, j},
   For[i = 1, i ≤ len, i ++,
    For[j = 1, j ≤ len, j ++,
     If[i == j, Continue[], If[m[[i, j]] =!= 0, Return[False]]]]];
   Return[True]];
```

We can now test the performance of this implementation :

```
diagonalQProc[testmatr] // Timing
diagonalQProc[testiden] // Timing
```

```
{5.34989 × 10^-15, False}
```

```
{2.263, True}
```

We see that it is very good on the matrix which is essentially non - diagonal, since with procedural approach we have the greatest flexibility to stop at any moment when the condition is violated. However, it is completely unsatisfactory for diagonal matrices, and we expect it to be also not great for very sparse matrices close to diagonal. Can we find a better all-around solution?

6.2.4 Functional implementations

This is an implementation based on MapIndexed (taken with a minor modification from the book of David Wagner. To avoid misunderstanding, let me add that he was not really interested in performance aspects in this example) :

```
Clear[diagonalQ];
diagonalQ[m_?MatrixQ] /; Equal @@ Dimensions[m] :=
  And @@ Flatten[MapIndexed[#1 == 0 || Equal @@ #2 &, m, {2}]];
```

Be sure to understand this code - it is a good warm-up. We now check the timing :

```
diagonalQ[testmatr] // Timing
diagonalQ[testiden] // Timing
```

```
{1.602, False}
```

```
{0.841, True}
```

We see that in general, it is even worse than the procedural version, because it is never fast. This is of course because MapIndexed traverses the entire matrix in any case, even when the result has already been established.

We can try to cure it by inserting Throw and Catch, so that the process stops right after the condition is first violated:

```
Clear[diagonalQNew];
diagonalQNew[m_?MatrixQ] /; Equal @@ Dimensions[m] :=
  If[# === False, #, True] &[Catch[MapIndexed[
     If[Not[#1 == 0 || Equal @@ #2], Throw[False]] &, m, {2}]]];
```

This helps somewhat in the first case, but even here not that much:

```
diagonalQNew[testmatr] // Timing
diagonalQNew[testiden] // Timing

{0.471, False}
{1.312, True}
```

6.2.5 Implementations based on structural operations

Now we will go a long way in trying to use certain functions optimized in *Mathematica*. The two main ideas will be to flatten the matrix and work with a flat list, and to vectorize our problem and use highly optimized vectorized operations.

The following implementation is a first step in this direction.

```
Clear[diagonalQ1];
diagonalQ1[m_?MatrixQ] /; Equal @@ Dimensions[m] := Total[
    Abs[Flatten[MapIndexed[Delete[#1, First[#2]] &, m]]]] === 0;
```

What happens here is that we use MapIndexed on level 1 to go through matrix rows and delete n - th element (the diagonal one) from each row. We then flatten the resulting matrix and sum all elements. If the matrix is diagonal, the sum should be zero.

```
diagonalQ1[testmatr] // Timing
diagonalQ1[testiden] // Timing

{0.26, False}
{0.06, True}
```

The results are certainly better than before.

The limitation of the approach based on summing all the elements is that the matrix elements can not be lists of different lengths.

This is the next logical step: instead of deleting diagonal elements from each row, we rotate each row so that the main diagonal becomes the first column, or the first row of the transposed matrix. Taking Rest of this matrix, flattening it and summing all the elements, we have to get zero if the matrix is diagonal.

```
Clear[diagonalQ2];
diagonalQ2[m_?MatrixQ] /; Equal @@ Dimensions[m] :=
  Total[Abs[Flatten[Rest[Transpose[
        MapIndexed[RotateLeft[#1, #2[[1]] - 1] &, m]]]]]] === 0;
```

We now check the performance :

```
diagonalQ2[testmatr] // Timing
diagonalQ2[testiden] // Timing
```

```
{0.261, False}
{0.07, True}
```

It is roughly the same as in our previous attempt. Can we get better?

Here we will delete the diagonal elements using MapThread, and then flatten the resulting matrix and compare it to a flat array of zeros of an appropriate length, which in this approach we have to generate :

```
Clear[diagonalQNew1];
diagonalQNew1[m_ ? MatrixQ] /; Equal @@ Dimensions[m] :=
 Module[{len = Length[m]},
  Flatten[MapThread[Delete, {m, Range[len]}]] ===
   Array[0 &, {len * (len -1)}]]
```

The performance of this version

```
diagonalQNew1[testmatr] // Timing
diagonalQNew1[testiden] // Timing
```

```
{0.17, False}
{0.02, True}
```

is certainly better than before.

Now we will develop the above idea further. Let us combine the two approaches: we will delete a diagonal element from the row, but then compare the resulting vector to a vector of zeros, using the high - performance Equal operator which is vectorized. We want to be able to stop after the first comparison yields False. As a first version, we may use Scan with a Return statement:

```
Clear[diagonalQNew20];
diagonalQNew20[m_ ? MatrixQ] /; Equal @@ Dimensions[m] :=
  Module[{len = Length[m], zeroarr},
   zeroarr = Array[0 &, {len -1}];
   Scan[If[Delete @@ # =!= zeroarr, Return[False]]  &,
    Transpose[{m, Range[len]}]];
   Return[True]];
```

Check :

```
diagonalQNew20[testmatr] // Timing
diagonalQNew20[testiden] // Timing
```

```
{0.01, True}
{0.041, True}
```

Similar functionality can be achieved with the use Fold in combination with Catch and Throw. This version seems a bit faster than the previous.

```
Clear[diagonalQNew21];
diagonalQNew21[m_ ? MatrixQ] /; Equal @@ Dimensions[m] :=
  Module[{len = Length[m], zeroarr},
   zeroarr = Array[0 &, {len -1}];
   Catch[Fold[If[Delete @@ #2 =!= zeroarr, Throw[False], True] &,
     False, Transpose[{m, Range[len]}]]]]];
```

What we have done in both cases is to first prepare a list of rows (the original matrix) supplied by the index of the diagonal element in every row, and then feed it to Scan or Fold. Let us check now :

```
diagonalQNew21[testmatr] // Timing
diagonalQNew21[testiden] // Timing
```

```
{0.01, False}
{0.02, True}
```

This is a quantum improvement. I have to say that this is more or less as good as it gets in terms of speed, and perhaps the best solution overall (see the comments below).

As the next step, let us first flatten the matrix, and create a list of positions of diagonal elements in the new flat list that our matrix became. Then, we use Delete to delete all these elements at once, and compare the resulting array to an array of zeros of an appropriate length :

```
Clear[diagonalQNew3];
diagonalQNew3[m_ ? MatrixQ] /; Equal @@ Dimensions[m] :=
  Module[{len = Length[m]},
   Delete[Flatten[m], List /@ Plus[Range[0, len -1] * (len +1),
      1]] === Array[0 &, {len * (len -1)}]];
```

The performance here,

```
diagonalQNew3[testmatr] // Timing
diagonalQNew3[testiden] // Timing
```

```
{0.01, False}
{0.02, True}
```

is not bad, but slightly worse than in our version with Fold, Catch and Throw. Can we use the same idea, but make it faster?

This is our last attempt. It is almost the same as before, but we don’ t generate the list of zeros. Rather, we sum all the elements (absolute values) and compare to zero.

```
Clear[diagonalQNew4];
diagonalQNew4[m_ ? MatrixQ] /; Equal @@ Dimensions[m] :=
  Module[{len = Length[m], poslist},
   poslist = List /@ Plus[Range[0, len - 1] * (len + 1), 1];
   Total[Abs[Delete[Flatten[m], poslist]]] === 0];
```

The performance now :

```
diagonalQNew4[testmatr] // Timing
diagonalQNew4[testiden] // Timing
```

```
{0.01, False}
{0.01, True}
```

The reason that this solution is faster than the previous one is that we need neither to generate a large array (of zeros), nor to compare it with our array. In fact, this solution (when applicable) beats all our previous ones, including the one with Fold, Catch and Throw (although not by a wide margin). Given the size of the matrices used, and the timing, this solution is acceptable.

There is one problem with this solution however - it may not work well on matrices of lists of different size, since Total can not add lists of different lengths, similarly to the situation with Plus and Subtract operations in the case study of section 5.3.2.3. We can not however solve this problem similarly to that case, without a loss in efficiency. This is because a lot of it has to do with our use of Total instead of Apply[Plus,...].

Thus, our best solutions are: <*diagonalQNew4*> for matrices whose elements are not lists of different lengths, and <*diagonalQNew20*> or <*diagonalQNew21*> in general case. Since their performance is almost the same, one will be safer picking the latter ones, which are cleaner in some sense.

6.2.6 Conclusions

I used this problem to illustrate several points already discussed in the text. In total, we have considered 10 different implementations, and the efficiency of the best and the worst one are several hundred times different. In implementing the solutions, we have used different programming styles and techniques. We have seen that just simply using functional programming instead of procedural helped us very little in this case - in fact, it made things worse. However, when we managed to combine functional programming with the heavily optimized vectorized operations such as vector comparisons, flattening the list structure or vector elements summing, the performance started to improve. The final solutions that we obtained are perhaps not as fast as the procedural one for the very dense non-diagonal matrices, but are more balanced and much better all-around. These final solutions are good enough to build any other application upon them and be sure that they will not be the cause of performance loss, (almost) like built-in functions.

The question is of course how "accidental" are these solutions: was it more a matter of luck or guess to obtain them, or were we "doomed" to get them at the end. My point is that it has nothing to do with luck. Admittedly, it takes some experience to make right guesses for the directions in which it is most promising to go, and also when it is the end and we have to stop because we won't do better. But the principles are always the same - try to work with as much data at once as possible, and prefer structural operations to anything else.

6.3 Case study II: extracting matrix diagonals

6.3.1 The problem

Consider the following problem: we need to extract from the matrix some or all of its diagonals, either right diagonals (going from top left to bottom right), or left diagonals (going from bottom left to top right). We should provide a list of matrix element positions, and our function has to extract all right or left (or both) diagonals which pass through these elements. And of course, we would like to do it as efficiently as possible.

6.3.2 Test matrices

```
(testmatr = Array[Random[Integer, {1, 15}] &, {4, 6}]) // MatrixForm
```

$$\begin{pmatrix} 6 & 7 & 14 & 1 & 8 & 6 \\ 11 & 13 & 15 & 6 & 12 & 11 \\ 12 & 4 & 5 & 11 & 7 & 4 \\ 10 & 8 & 10 & 8 & 14 & 11 \end{pmatrix}$$

```
powertestmatr = Array[Random[Integer, {1, 50}] &, {500, 500}];
```

6.3.3 Extract - based implementation

6.3.3.1 The implementation

The idea of this implementation is to generate a list of positions for elements on each diagonal that we need to extract. Then, we can use the built - in Extract, which may accept a list of positions of elements to be extracted.

Since we define a diagonal by any element through which it passes, we will need a number of auxiliary functions.

First, we will need a function which takes matrix dimensions and an address of a single element < elem > in question, and determines the address of the "starting" element of (say, right) diagonal which passes through < elem > . This is done by the following code :

```
Clear[diagRightStart];
diagRightStart[{r_Integer ? Positive, c_Integer ? Positive},
    {rows_Integer ? Positive, columns_Integer ? Positive}]  /;
   r ≤ rows && c ≤ columns :=
  Switch[r - c, _ ? Positive, {1 + (r - c), 1},
   _ ? Negative, {1, 1 + c - r}, 0, {1, 1}
  ];

diagRightStart[x__] := {};
```

Here, < r > and < c > represent a row and column of the element in question, while < rows > and < columns > represent matrix dimensions. For example :

```
diagRightStart[{4, 3}, {5, 5}]
```

```
{2, 1}
```

Next we will need a function which generalizes the above to a list of elements (positions). Basically, it has to Map the previous function on a list of element positions. Here is the code :

```
Clear[startRightDiagPositions];
startRightDiagPositions[{rows_Integer ? Positive,
    columns_Integer ? Positive}, {elements__List}] :=
  DeleteCases[Map[diagRightStart[#, {rows, columns}] &,
    {elements}], {}];
```

For example,

```
startRightDiagPositions[{5, 5}, {{2, 3}, {7, 8}, {3, 3}, {3, 4}}]
```

```
{{1, 2}, {1, 1}, {1, 2}}
```

We see that elements with indices out of range, are ignored (if this is not the desired behavior, this can be easily changed). At the same time, for several elements on a same diagonal, its starting address is repeated as many times as is the number of elements on this diagonal among the input elements (in other words, we do not eliminate multiple references to the same diagonal, and in general it may be extracted more than once).

We may need eventually to extract all right or all left diagonals (or may be, both). The two auxiliary functions below provide the "starting" addresses of elements for these diagonals - we will use this later.

```
Clear[allRightDiagStartPositions];
allRightDiagStartPositions[
   {rows_Integer ? Positive, columns_Integer ? Positive}] :=
  Join[Thread[{Range[rows, 2, -1], 1}], {{1, 1}},
   Thread[{1, Range[2, columns]}]];
```

```
Clear[allLeftDiagStartPositions];
allLeftDiagStartPositions[
   {rows_Integer ? Positive, columns_Integer ? Positive}] :=
  Join[Thread[{Range[rows], 1}],
   Thread[{rows, Range[2, columns]}]];
```

For example :

```
allRightDiagStartPositions[{5, 7}]
```

```
{{5, 1}, {4, 1}, {3, 1}, {2, 1}, {1, 1},
 {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}}
```

```
allLeftDiagStartPositions[{5, 7}]
```

```
{{1, 1}, {2, 1}, {3, 1}, {4, 1}, {5, 1},
 {5, 2}, {5, 3}, {5, 4}, {5, 5}, {5, 6}, {5, 7}}
```

Once the "starting" positions of the diagonals are calculated, we need to generate full lists of positions of diagonal elements, to be used in Extract. The following function will do it for a single right diagonal (we don’t need a separate one for the left diagonal, as we will see):

```
Clear[rightDiagPositions];
rightDiagPositions[{1, col_Integer ? Positive},
   {rows_Integer ? Positive, columns_Integer ? Positive}]  :=
  With[{len = Min[rows, columns - col + 1]},
   Transpose[{Range[len], Range[col, col + len - 1]}]];

rightDiagPositions[{row_Integer ? Positive, 1},
   {rows_Integer ? Positive, columns_Integer ? Positive}]  :=
  With[{len = Min[columns, rows - row + 1]},
   Transpose[{Range[row, row + len - 1], Range[len]}]];
```

It uses the fact that the right diagonal always starts with an element either in the first row or in the first column (or both, for {1, 1} element). For example :

```
rightDiagPositions[{2, 1}, {4, 6}]
{{2, 1}, {3, 2}, {4, 3}}
rightDiagPositions[{1, 3}, {4, 6}]
{{1, 3}, {2, 4}, {3, 5}, {4, 6}}
```

Now we are ready to write our main function. It will take a matrix and a list of elements, process the element positions by the above auxiliary functions, generate position lists for elements of the diagonals, and then Map Extract on these position lists to extract the diagonals :

```
Clear[getSomeRightMatrixDiagonals];
getSomeRightMatrixDiagonals[
   matr_ ? MatrixQ, els : ({elements__List} | All)] :=
  With[{dims = Dimensions[matr]},
   Module[{poslist},
    poslist =
     If[els === All, allRightDiagStartPositions[dims],
      startRightDiagPositions[dims, {elements}]
     ];
    Map[Extract[matr,
       rightDiagPositions[#, dims]] &, poslist]
   ]
  ];
```

Notice the alternative pattern and named pattern used to incorporate the case where we need all of the diagonals. The symbol < All > is a system symbol used in such cases as ours, so we don’ t need to invent a new one. Notice that when we use All, a list of elements is generated by one of our auxiliary functions <allRightDiagStartPositions>, discussed earlier. Examples:

```
testmatr // MatrixForm
```

```
 6   7  14  1   8   6
11  13  15  6  12  11
12   4   5 11   7   4
10   8  10  8  14  11
```

```
getSomeRightMatrixDiagonals[testmatr,
 {{1, 1}, {1, 3}, {1, 4}, {3, 1}, {5, 1}, {3, 3}}]
```

```
{{6, 13, 5, 8}, {14, 6, 7, 11}, {1, 12, 4}, {12, 8}, {6, 13, 5, 8}}
```

```
getSomeRightMatrixDiagonals[testmatr, All]
```

```
{{10}, {12, 8}, {11, 4, 10}, {6, 13, 5, 8},
 {7, 15, 11, 14}, {14, 6, 7, 11}, {1, 12, 4}, {8, 11}, {6}}
```

Notice that if there is more than one element belonging to the same diagonal, this diagonal is extracted more than once. If this is not the desired behavior, one can introduce an option which will control it, by effectively applying something like UnsortedUnion to the list of diagonal starting positions. This is left as an exercise to the reader.

We only solved a problem for right diagonals. For the left diagonals, we in fact don' t need to solve it again, if we realize that by reversing the order of rows in our matrix, and manipulating the positions of the elements in the element list accordingly, we can use our right - diagonal solution. Here is the code :

```
Clear[getSomeLeftMatrixDiagonals];
getSomeLeftMatrixDiagonals[matr_ ? MatrixQ,
   els : ({elements__List} | All)] := Module[{rows , columns, elems},
   {rows, columns}  = Dimensions[matr];
   elems =
    If[els === All,
     allLeftDiagStartPositions[{rows, columns}],
     {elements}
    ];
   getSomeRightMatrixDiagonals[Reverse[matr],
    Transpose[MapAt[rows +1 -# &, Transpose[elems], 1]]]
  ];
```

The double Transpose on the element position list is needed to do the transformation only on the row - component of each element' s position. As before, if we use <All> rather than provide an explicit element list, a list of elements for all the left diagonals is generated. Examples:

```
getSomeLeftMatrixDiagonals[testmatr,
 {{1, 1}, {1, 3}, {1, 4}, {3, 1}, {5, 1}}]
```

```
{{6}, {12, 13, 14}, {10, 4, 15, 1}, {12, 13, 14}}
```

```
getSomeLeftMatrixDiagonals[testmatr, All]
```

```
{{6}, {11, 7}, {12, 13, 14}, {10, 4, 15, 1},
 {8, 5, 6, 8}, {10, 11, 12, 6}, {8, 7, 11}, {14, 4}, {11}}
```

Finally, we may get both right and left diagonals passing through each of the elements from our element list. This is done by the following code :

```
Clear[getSomeMatrixDiagonals];
getSomeMatrixDiagonals[
   matr_?MatrixQ, els : ({elements__List} | All)] :=
  Transpose[{getSomeLeftMatrixDiagonals[matr, els],
    getSomeRightMatrixDiagonals[matr, els]}];
```

The use of Transpose ensures that the diagonals are grouped in pairs, according to the elements. Examples:

```
getSomeMatrixDiagonals[testmatr,
 {{1, 1}, {1, 3}, {1, 4}, {3, 1}, {5, 1}}]
```

```
{{{6}, {6, 13, 5, 8}}, {{12, 13, 14}, {14, 6, 7, 11}},
 {{10, 4, 15, 1}, {1, 12, 4}}, {{12, 13, 14}, {12, 8}}}
```

```
getSomeMatrixDiagonals[testmatr, All]
```

```
{{{6}, {10}}, {{11, 7}, {12, 8}},
 {{12, 13, 14}, {11, 4, 10}}, {{10, 4, 15, 1}, {6, 13, 5, 8}},
 {{8, 5, 6, 8}, {7, 15, 11, 14}}, {{10, 11, 12, 6}, {14, 6, 7, 11}},
 {{8, 7, 11}, {1, 12, 4}}, {{14, 4}, {8, 11}}, {{11}, {6}}}
```

In the case when we use <All>, the grouping of diagonals does not have a direct meaning. If we Transpose the result in this case, we will get a list of two sublists, containing left and right diagonals in the right order.

- 6.3.3.2 Performance tests

This measures time needed to extract all left, all right, and left + right matrix diagonals.

```
getSomeRightMatrixDiagonals[powertestmatr, All] // myTiming
```

```
0.0792
```

```
getSomeLeftMatrixDiagonals[powertestmatr, All] // Short // Timing
```

```
{0.09, {{4}, {20, 15}, <<996>>, {30}}}
```

```
getSomeMatrixDiagonals[powertestmatr, All] // Short // Timing
```

```
{0.19, {{{4}, {5}}, <<997>>, <<1>>}}
```

The timings are not bad for this matrix size (500 x 500)

If we need just some diagonals, then the time needed will of course be much less. Here we extract 3 left and 3 right diagonals, passing through the elements with the positions below:

```
getSomeMatrixDiagonals[powertestmatr,
  {{1, 1}, {5, 10}, {3, 7}}] // myTiming
```

```
0.0042
```

■ 6.3.3.3 Appendix : the complete code of the Extract - based implementation

Here we simply assemble all the functions in one place

```
Clear[diagRightStart];
diagRightStart[{r_Integer ? Positive, c_Integer ? Positive},
    {rows_Integer ? Positive, columns_Integer ? Positive}] /;
   r ≤ rows && c ≤ columns :=
  Switch[r - c,
   _ ? Positive, {1 + (r - c), 1},
   _ ? Negative, {1, 1 + c - r}, 0, {1, 1}
  ];

diagRightStart[x__] := {};


Clear[startRightDiagPositions];
startRightDiagPositions[{rows_Integer ? Positive,
    columns_Integer ? Positive}, {elements__List}] :=
  DeleteCases[
   Map[diagRightStart[#, {rows, columns}] &, {elements}], {}
  ];


Clear[allRightDiagStartPositions];
allRightDiagStartPositions[
   {rows_Integer ? Positive, columns_Integer ? Positive}] :=
  Join[Thread[{Range[rows, 2, -1], 1}], {{1, 1}},
   Thread[{1, Range[2, columns]}]];


Clear[allLeftDiagStartPositions];
allLeftDiagStartPositions[
   {rows_Integer ? Positive, columns_Integer ? Positive}] :=
  Join[Thread[{Range[rows], 1}],
   Thread[{rows, Range[2, columns]}]
  ];
```

```
Clear[rightDiagPositions];
rightDiagPositions[{1, col_Integer ? Positive},
   {rows_Integer ? Positive, columns_Integer ? Positive}] :=
  With[{len = Min[rows, columns - col + 1]},
   Transpose[{Range[len], Range[col, col + len - 1]}]
  ];


rightDiagPositions[{row_Integer ? Positive, 1},
   {rows_Integer ? Positive, columns_Integer ? Positive}] :=
  With[{len = Min[columns, rows - row + 1]},
   Transpose[{Range[row, row + len - 1], Range[len]}]
  ];


Clear[getSomeRightMatrixDiagonals];
getSomeRightMatrixDiagonals[
   matr_ ? MatrixQ, els : ({elements__List} | All)] :=
  With[{dims = Dimensions[matr]},
   Module[{poslist},
    poslist =
     If[els === All, allRightDiagStartPositions[dims],
      startRightDiagPositions[dims, {elements}]
     ];
    Map[Extract[matr, rightDiagPositions[#, dims]] &, poslist]
   ]
  ];


Clear[getSomeLeftMatrixDiagonals];
getSomeLeftMatrixDiagonals[
   matr_ ? MatrixQ, els : ({elements__List} | All)] :=
  Module[{rows , columns, elems},
   {rows, columns} = Dimensions[matr];
   elems =
    If[els === All,
     allLeftDiagStartPositions[{rows, columns}], {elements}
    ];
   getSomeRightMatrixDiagonals[Reverse[matr],
    Transpose[MapAt[rows + 1 - # &, Transpose[elems], 1]]]
  ];
```

```
Clear[getSomeMatrixDiagonals];
getSomeMatrixDiagonals[
   matr_ ? MatrixQ, els : ({elements__List} | All)] :=
  Transpose[{getSomeLeftMatrixDiagonals[matr, els],
    getSomeRightMatrixDiagonals[matr, els]}];
```

■ 6.3.4 Procedural implementation

■ 6.3.4.1 The implementation

In fact, the only place where the procedural implementation will be different is the extraction of diagonals proper. It will use several of the auxiliary functions that we have developed for the Extract - based version. The idea will be to first process a list of element positions to get a list of starting positions for the diagonals, and then extract the diagonals. In a procedural version, first we have to write a function which extracts a single diagonal:

```
Clear[extractSingleRightDiagProc];
extractSingleRightDiagProc[matr_ ? MatrixQ,
   {row_Integer ? Positive, col_Integer ? Positive}] :=
  Module[{rows, columns, i, j, k, startpos, result},
   {rows, columns}  = Dimensions[matr];
   result = Array[0 &, {Max[rows, columns]}];
   If[Head[startpos =
       diagRightStart[{row, col}, {rows, columns}]]  =!= List,
    Return[{}],
    {i, j} = startpos
   ];
   For[k = 0, (i + k ≤ rows) && (j + k ≤ columns), k ++,
    result[[k + 1]] = matr[[i + k, j + k]];
   ];
   result = Take[result, k]
  ];
```

For example :

```
testmatr // MatrixForm
```

$$\begin{pmatrix} 6 & 7 & 14 & 1 & 8 & 6 \\ 11 & 13 & 15 & 6 & 12 & 11 \\ 12 & 4 & 5 & 11 & 7 & 4 \\ 10 & 8 & 10 & 8 & 14 & 11 \end{pmatrix}$$

```
extractSingleRightDiagProc[testmatr, {1, 1}]
```

```
{6, 13, 5, 8}
```

```
extractSingleRightDiagProc[testmatr, {4, 1}]
```

```
{10}
```

Now we have to write a function which will extract many or all of the diagonals:

```
Clear[extractSomeRightDiagsProc];
extractSomeRightDiagsProc[
   matr_?MatrixQ, els : ({elements__List} | All)] :=
  With[{dims = Dimensions[matr]},
   Module[{poslist, i, len, result},
    poslist =
     If[els === All, allRightDiagStartPositions[dims],
      startRightDiagPositions[dims, {elements}]
     ];
    len = Length[poslist];
    result = Table[0, {len}];
    For[i = 1, i ≤ len, i ++,
     result[[i]] = extractSingleRightDiagProc[matr, poslist[[i]]]
    ];
    result
   ]
  ];
```

Check :

```
extractSomeRightDiagsProc[testmatr,
 {{1, 1}, {1, 3}, {1, 4}, {3, 1}, {5, 1}, {3, 3}}]
{{6, 13, 5, 8}, {14, 6, 7, 11}, {1, 12, 4}, {12, 8}, {6, 13, 5, 8}}
extractSomeRightDiagsProc[testmatr, All]
{{10}, {12, 8}, {11, 4, 10}, {6, 13, 5, 8},
 {7, 15, 11, 14}, {14, 6, 7, 11}, {1, 12, 4}, {8, 11}, {6}}
```

The functions to extract left diagonals and all diagonals are exactly the same as before, modulo changing the name of the function which extracts right diagonals from < getSomeRightMatrixDiagonals > to < extractSomeRightDiagsProc > .

- 6.3.4.2 Performance tests

```
extractSomeRightDiagsProc[powertestmatr,
   {{1, 1}, {3, 7}, {5, 10}}] // Short // Timing
{0.06, {<<1>>}}
extractSomeRightDiagsProc[powertestmatr, All] // Short // Timing
{6.119, {{5}, {47, 18}, <<996>>, {12}}}
```

As we can see, the procedural version is hopelessly slow, even though we did not use Append and Prepend in list accumulation, and in fact used the same auxiliary functions for the preliminary steps. Thus, the comparison is rather "clean" and fair. For the present example, the difference in performance is about 100 times! And pretty much this difference remains even when I only have to extract a few diagonals. Had I replaced all the code in those auxiliary functions also by its purely procedural version, the difference would have been even more dramatic.

I replaced all the code in those auxilliary functions also by its purely procedural version, the difference would have been even more dramatic.

- 6.3.5 The fastest version for all diagonal extraction, based on structural operations

- 6.3.5.1 Developing a solution

While the above Extract - based implementation is our best all - round one, we could win another factor of 1.5~2 in performance in extracting all matrix diagonals (or, all left, all right ones) by using the structural operations. Let me briefly demonstrate how such code may be developed.

```
testmatr // MatrixForm
```

$$\begin{pmatrix} 6 & 7 & 14 & 1 & 8 & 6 \\ 11 & 13 & 15 & 6 & 12 & 11 \\ 12 & 4 & 5 & 11 & 7 & 4 \\ 10 & 8 & 10 & 8 & 14 & 11 \end{pmatrix}$$

We first find number of rows and columns :

```
{testrows, testcolomns} = Dimensions[testmatr]
```

```
{4, 6}
```

We will now use RotateLeft and MapThread to rotate individual rows differently (similar to one of the examples on MapThread, section 5.3.2.4.5) :

```
step1 =
 Transpose[MapThread[RotateLeft, {testmatr, Range[0, testrows - 1]}]]
```

```
{{6, 13, 5, 8}, {7, 15, 11, 14}, {14, 6, 7, 11},
 {1, 12, 4, 10}, {8, 11, 12, 8}, {6, 11, 4, 10}}
```

If you look carefully at the sublists, you will see that they either represent full left diagonals (this will be so for the first (columns - rows + 1) sublist), or they represent 2 right diagonals glued together. We have then to "unglue" them. This is how it is done:

```
step2 = MapThread[{Drop[#1, -#2], Take[#1, -#2]} &,
  {step1, PadLeft[Range[0, testrows - 1], testcolomns]}]
```

```
{{{6, 13, 5, 8}, {}}, {{7, 15, 11, 14}, {}}, {{14, 6, 7, 11}, {}},
 {{1, 12, 4}, {10}}, {{8, 11}, {12, 8}}, {{6}, {11, 4, 10}}}
```

We have used Padleft to account for the first several diagonals which are complete. In this case, we use MapThread in a similar spirit as before, but this time with a {Drop[#1, -#2], Take[#1, -#2]} & function.

Now, we have in principle already obtained all right diagonals, but they are grouped unnaturally - they are not in any simple logical order. We have to reorder them. This turns out to be easy to do:

```
step3 = Flatten[Reverse[Transpose[step2]], 1]
```

```
{{}, {}, {}, {10}, {12, 8}, {11, 4, 10}, {6, 13, 5, 8},
 {7, 15, 11, 14}, {14, 6, 7, 11}, {1, 12, 4}, {8, 11}, {6}}
```

We now just have to delete the empty lists from the beginning. We know precisely how many there are : (columns - rows + 1) of them, since they result from the complete diagonals. We have then to Drop them:

```
result = Drop[step3, testcolomns - testrows + 1]
{{10}, {12, 8}, {11, 4, 10}, {6, 13, 5, 8},
 {7, 15, 11, 14}, {14, 6, 7, 11}, {1, 12, 4}, {8, 11}, {6}}
```

These are the diagonals in the correct order - starting from the smallest (bottom - left) to larger ones and again smaller (top - right). We can now combine the steps into a function:

```
Clear[matrixRightDiagonals]
matrixRightDiagonals[matr_ ? MatrixQ] /;
   NonPositive[Subtract @@ Dimensions[matr]] :=
  Module[{rows, columns},
   {rows, columns}  = Dimensions[matr];
   Drop[
    Flatten[
     Reverse[
      Transpose[
       MapThread[{Drop[#1, -#2], Take[#1, -#2]} &,
        {Transpose[
          MapThread[RotateLeft,
           {matr, Range[0, rows - 1]}
          ]
         ],
         PadLeft[Range[0, rows - 1], columns]
        }
       ](* MapThread *)
      ](* Transpose *)
     ], (* Reverse *)
     1], (* Flatten *)
    columns - rows + 1]  (*Drop *)
  ];
```

Check :

```
matrixRightDiagonals[testmatr]
{{10}, {12, 8}, {11, 4, 10}, {6, 13, 5, 8},
 {7, 15, 11, 14}, {14, 6, 7, 11}, {1, 12, 4}, {8, 11}, {6}}
```

For matrices with the number of rows larger than the number of columns, it is enough to consider the Transpose - d matrix and then reverse the resulting list of diagonals :

```
matrixRightDiagonals[matr_ ? MatrixQ] /;
   Positive[Subtract @@ Dimensions[matr]] :=
  Reverse[matrixRightDiagonals[Transpose[matr]]];
```

Check :

```
Transpose[testmatr] // MatrixForm
```

```
⎛ 6  11 12 10⎞
⎜ 7  13  4  8⎟
⎜14  15  5 10⎟
⎜ 1   6 11  8⎟
⎜ 8  12  7 14⎟
⎝ 6  11  4 11⎠
```

```
matrixRightDiagonals[Transpose[testmatr]]
```

```
{{6}, {8, 11}, {1, 12, 4}, {14, 6, 7, 11},
 {7, 15, 11, 14}, {6, 13, 5, 8}, {11, 4, 10}, {12, 8}, {10}}
```

For the left diagonals, as we already saw, it is enough to reverse the matrix and then consider the right diagonals once again :

```
Clear[matrixLeftDiagonals];
matrixLeftDiagonals[matr_ ? MatrixQ] :=
  matrixRightDiagonals[Reverse[matr]];
```

Check :

```
testmatr // MatrixForm
```

```
⎛ 6   7 14  1  8  6⎞
⎜11  13 15  6 12 11⎟
⎜12   4  5 11  7  4⎟
⎝10   8 10  8 14 11⎠
```

```
matrixLeftDiagonals[testmatr]
```

```
{{6}, {11, 7}, {12, 13, 14}, {10, 4, 15, 1},
 {8, 5, 6, 8}, {10, 11, 12, 6}, {8, 7, 11}, {14, 4}, {11}}
```

Finally, the function for all the diagonals is written straightforwardly :

```
Clear[getAllMatrixDiagonals];
getAllMatrixDiagonals[matr_ ? MatrixQ] :=
  Through[{matrixRightDiagonals, matrixLeftDiagonals}[matr]];
```

Check :

```
getAllMatrixDiagonals[testmatr]
```

```
{{{10}, {12, 8}, {11, 4, 10}, {6, 13, 5, 8},
  {7, 15, 11, 14}, {14, 6, 7, 11}, {1, 12, 4}, {8, 11}, {6}},
 {{6}, {11, 7}, {12, 13, 14}, {10, 4, 15, 1}, {8, 5, 6, 8},
  {10, 11, 12, 6}, {8, 7, 11}, {14, 4}, {11}}}
```

■ 6.3.5.2 Performance tests

We can now test the performance of new functions on our power example :

```
matrixRightDiagonals[powertestmatr]; // myTiming
```

```
0.0611
matrixLeftDiagonals[powertestmatr]; // myTiming
0.0611
getAllMatrixDiagonals[powertestmatr]; // myTiming
0.13
```

We see that we get a factor of 1.5-2 performance gain with respect to the Extract - based implementation, but of course the present one is only limited to the case when we need all the diagonals. And because it performs certain operations such as RotateLeft on the entire matrix, we can not hope for an easy optimization of the present solution to the case of just a few matrix diagonals.

6.3.5.3 Appendix : the complete code for the structural solution

```
Clear[matrixRightDiagonals]
matrixRightDiagonals[matr_ ? MatrixQ] /;
   NonPositive[Subtract @@ Dimensions[matr]] :=
  Module[{rows, columns},
   {rows, columns} = Dimensions[matr];
   Drop[Flatten[
     Reverse[Transpose[MapThread[{Drop[#1, -#2], Take[#1, -#2]} &,
        {Transpose[MapThread[RotateLeft, {matr, Range[0, rows - 1]}]],
         PadLeft[Range[0, rows - 1],
          columns]}]]]], 1], columns - rows + 1]];

matrixRightDiagonals[matr_ ? MatrixQ] /;
   Positive[Subtract @@ Dimensions[matr]] :=
  Reverse[matrixRightDiagonals[Transpose[matr]]];

Clear[matrixLeftDiagonals];
matrixLeftDiagonals[matr_ ? MatrixQ] :=
  matrixRightDiagonals[Reverse[matr]];

Clear[getAllMatrixDiagonals];
getAllMatrixDiagonals[matr_ ? MatrixQ] :=
  Through[{matrixRightDiagonals, matrixLeftDiagonals}[matr]];
```

6.3.6 Conclusions

The problem in question is a good playground to see how different programming styles and techniques compare to each other in terms of speed, flexibility etc. We have seen that the implementation based on functional programming and Extract operator is a very fast, good all-round solution. We also saw that the procedural solution is hopelessly slow, even though we were always careful to pre-allocate the result and not use Append etc. Finally, we have seen that in case when we need all the diagonals, there exists another solution based on structural operations, which is even faster than the Extract-based one, but limited to the case of all diagonals only.

Another part of this problem which can be a good exercise and also would add a practical value to it, is to efficiently modify or replace given matrix diagonals, not just extract them. It should be possible to tackle it by a similar method, using ReplacePart rather than Extract (however, let us keep in mind that ReplacePart has efficiency issues for large number of simultaneous replacements).

6.4 Case study III: generating complex random Wishart matrices

▪ 6.4.1 The problem

Here we will consider a problem of generation of complex random Wishart matrices with the normally distributed complex entries. Wishart matrices are block matrices of the form

```
{{0, W}, {WH, 0}} // MatrixForm
```

$$\begin{pmatrix} 0 & W \\ WH & 0 \end{pmatrix}$$

where W is n x m complex matrix and WH is its hermitian conjugate. Thus, the total matrix is (m + n) x (m + n). Random complex Wishart matrices with uncorrelated (other than due to WH being hermitian conjugate of W) normally distributed entries of W, form the so-called chiral ensembles and are used, in particular, for the low-energy description of Quantum ChromoDynamics. Of main interest usually are the eigenvalue correlations, but here we will just consider a way to generate these matrices.

▪ 6.4.2 Preliminaries

We will need to load a package Statistics‘ContinuousDistributions :

```
<<Statistics‘ContinuousDistributions‘
```

The way to generate the gaussian numbers is as follows :

```
RandomArray[NormalDistribution[0, 1], {10}]
{-0.284087, -0.364616, -0.0643195, -2.03162,
 -1.03452, 1.74043, -0.13117, 2.22543, -3.16775, 1.32895}
```

Here we have generated 10 random numbers with zero mean and unit variance.

▪ 6.4.3 Procedural implementation

Here is the straightforward procedural implementation. This function generates a single Wishart matrix of specified dimensions, mean and variance of the elements distribution.

```
Clear[buildMatrixProc];
buildMatrixProc[m_Integer,
   n_Integer, mu_ ? NumericQ, sigma_ ? NumericQ] :=
  Module[{source , nums, i, j, k = 1,
    result = Array[0 &, {m + n, m + n}]},
   source = RandomArray[NormalDistribution[mu, sigma], {2 m n}];
   nums = Take[source, n m] + I * Drop[source, m n];
   For [i = 1, i ≤ n, i ++,
    For[j = n + 1, j ≤ n + m, j ++,
     result[[j, i]] =
       Conjugate[
        result[[i, j]] = nums[[k ++]]
       ];
    ]
   ];
   result
  ];
```

It does it by preallocating a matrix of (n + m) x (n + m) zeros, generating n*m complex numbers and then using a nested loop to insert these numbers (or complex conjugate) into the place where W (WH) should be. For example :

```
buildMatrixProc[1, 2, 0, 1]
{{0, 0, -1.21174 - 0.0416826 ⅈ}, {0, 0, 0.983883 + 0.387592 ⅈ},
 {-1.21174 + 0.0416826 ⅈ, 0.983883 - 0.387592 ⅈ, 0}}
```

Let us see how long it will take to produce 1000 matrices with n = m = 10 (that is, 20 x 20 matrices) :

```
Do[buildMatrixProc[10, 10, 0, 1], {1000}]; // Timing
{6.96, Null}
```

■ 6.4.4 Functional implementation

Let us try to improve performance by creating blocks W and WH with the use of Partition command, and then just joining 4 blocks : 2 blocks of zeros, W and WH.

We will need an auxiliary function to join 4 submatrices into a matrix. For example, for the blocks :

```
MatrixForm /@ {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}},
  {{9, 10}, {11, 12}}, {{13, 14}, {15, 16}}}
```

$$\left\{\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}, \begin{pmatrix}5 & 6\\7 & 8\end{pmatrix}, \begin{pmatrix}9 & 10\\11 & 12\end{pmatrix}, \begin{pmatrix}13 & 14\\15 & 16\end{pmatrix}\right\}$$

We have to get

```
{{1, 2, 5, 6}, {3, 4, 7, 8},
  {9, 10, 13, 14}, {11, 12, 15, 16}} // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{pmatrix}$$

Here is the code :

```
Clear[join4Blocks];
join4Blocks[matrA_List, matrB_List, matrC_List, matrD_List] :=
  Flatten[Map[Join @@@ Transpose[#] &,
    {{matrA, matrB}, {matrC, matrD}}], 1];
```

Check :

```
join4Blocks[{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}},
 {{9, 10}, {11, 12}}, {{13, 14}, {15, 16}}]
```

```
{{1, 2, 5, 6}, {3, 4, 7, 8}, {9, 10, 13, 14}, {11, 12, 15, 16}}
```

I leave it as an exercise to the reader to figure out how this function works.

Now, we are ready to write our functional version for Wishart matrices generation :

```
Clear[buildWishartMatrix];
buildWishartMatrix[n_Integer,
   m_Integer, mu_ ? NumericQ, sigma_ ? NumericQ] :=
  Module[{source = RandomArray[NormalDistribution[mu, sigma],
      {2 m n}], wmatrix, nums},
   nums = Take[source, n m] + I * Drop[source, m n];
   wmatrix = Partition[nums, m];
   join4Blocks[Array[0 &, {n, n}], wmatrix,
    Conjugate[Transpose[wmatrix]], Array[0 &, {m, m}]]
  ];
```

Its first part is the same as before. Then instead of using loops we use Partition to create the W matrix, and then use the joining function above, to build the Wishart matrix.For example :

```
buildWishartMatrix[2, 1, 0, 1]
```

```
{{0, 0, -0.881925 + 0.951021 ⅈ}, {0, 0, 1.06796 - 1.08175 ⅈ},
 {-0.881925 - 0.951021 ⅈ, 1.06796 + 1.08175 ⅈ, 0}}
```

Let us see if we gained anything :

```
Do[buildWishartMatrix[10, 10, 0, 1], {1000}]; // Timing
```

```
{4.156, Null}
```

We see that we get about 70 - 100 % increase in speed, with respect to the procedural solution.

6.4.5 Implementation based on structural operations

Following our main rule of thumb, which states that we must use as large piece of data as possible at a time, let us ask ourselves : is it possible to create all the matrices we need at once, without a separate function which builds a single matrix? This sounds rather strange, but the answer is yes. Here is the implementation:

```
Clear[recmatrices];
recmatrices[vertsize_, horsize_, matnum_, data_] :=
Module[{initpartition, upper, lower},
initpartition = Partition[data, horsize];
upper = Partition[Flatten[#, 1] & /@
      (Join[{Partition[Array[0 &, {vertsize^2 * matnum}],
            vertsize]}, {initpartition}] // Transpose), vertsize];
lower = Partition[Flatten[#, 1] & /@ Transpose[Join[{Flatten[
           Transpose /@ Partition[initpartition, vertsize], 1]},
         {Partition[Array[0 &, {horsize^2 * matnum}],
           horsize]}]], horsize];
Flatten[#, 1] & /@ Transpose[Join[{upper}, {Conjugate[lower]}]]
  ];
```

What it does is to build lower and upper parts of all matrices we need at once, by the liberal use of Partition and Transpose commands. The result is a list of matrices, but it is not prepared matrix by matrix. Rather, the structural manipulations affect all matrices at once. It takes vertical size (n), horizontal size (m), total number of matrices needed, and a list of data to fill the matrices with (complex random numbers in our case).To illustrate the way it works, consider :

```
MatrixForm /@ recmatrices[2, 3, 5, Range[30]]
```

$$\left\{\begin{pmatrix}0&0&1&2&3\\0&0&4&5&6\\1&4&0&0&0\\2&5&0&0&0\\3&6&0&0&0\end{pmatrix},\begin{pmatrix}0&0&7&8&9\\0&0&10&11&12\\7&10&0&0&0\\8&11&0&0&0\\9&12&0&0&0\end{pmatrix},\begin{pmatrix}0&0&13&14&15\\0&0&16&17&18\\13&16&0&0&0\\14&17&0&0&0\\15&18&0&0&0\end{pmatrix},\begin{pmatrix}0&0&19&20&21\\0&0&22&23&24\\19&22&0&0&0\\20&23&0&0&0\\21&24&0&0&0\end{pmatrix},\begin{pmatrix}0&0&25&26&27\\0&0&28&29&30\\25&28&0&0&0\\26&29&0&0&0\\27&30&0&0&0\end{pmatrix}\right\}$$

Basically, it first prepares the upper part of all matrices at once, and then the lower one for all matrices at once, in such way that the non - zero lower block WH is exactly the hermitian conjugate of W for each matrix. It is not difficult to dissect the code and see what is going on.

For this version, we will need an external function to prepare a list of random complex numbers for us :

```
Clear[prepareComplexNormal];
prepareComplexNormal[
   mu_ ? NumericQ, sigma_ ? NumericQ, n_Integer] :=
  Plus[Take[#, Length[#] / 2], I * Drop[#, Length[#] / 2]] &[
   RandomArray[NormalDistribution[mu, sigma], {2 n}]];
```

Now we can test the performance :

```
ourmatrices  = recmatrices[10, 10, 1000,
    prepareComplexNormal[0, 1, 100 000]]; // Timing
{1.201, Null}
```

We see that we get another factor of 3 improvement as compared to the functional version.

Let us see how long it takes the built - in Eigenvalues to diagonalize these matrices :

```
Eigenvalues /@ ourmatrices; // Timing
{1.472, Null}
```

We see that it is about the same time as needed for their generation. While ideally we should be able to generate the matrices much faster than to diagonalize them, this is marginally acceptable. The situation with a procedural version where it takes 5 times longer to generate the matrix than to diagonalize it, is not.

6.4.6 Conclusions

This problem is just another example to illustrate the relative efficiency of different programming styles. The performance difference between the fastest (structural) solution and the slowest (procedural) one is not as dramatic here as in other examples in this chapter and is about 5 - 6 times, but it is nevertheless substantial. It means for example that I can gather the same statistics with the structural solution in 1 hour as with the procedural solution in 5 - 6 hours.

As before, we have seen that the procedural style is the least efficient, functional one is in between, and the one based on the structural operations is the fastest by a wide margin. Admittedly, however, it requires a rather counter-intuitive thinking, which may be considered a disadvantage in terms of program readability and maintenance. However, if the gain is a 3-fold performance increase, it may be worth it.

6.5 Case study IV: sorting, mapping and membership tests

6.5.1 The problem

It is often needed in practice to do something to those elements of one list (set) which are also the members of another list (set). Some examples of this kind we have considered before - the unsorted Intersection function (section 4.3.3.4), and unsorted Union function (section 5.2.6.2.5), but this is a rather general formulation. Here we will consider a following problem: we need to Map some function <f> on these elements. As usual, we will move from the easiest procedural solution to more efficient ones, and develop some generally useful functionality along the way.

6.5.2 Test sets

These will be our simple test sets that we will use to develop our functions: a first list on which elements we should Map f, and the second list membership in which we will test for the elements of the first one.

```
Clear[f];
```

```
testlst = Table[Random[Integer, {1, 15}], {20}]
```

```
{1, 5, 3, 6, 7, 2, 4, 2, 8, 14, 12, 13, 4, 14, 2, 4, 4, 13, 9, 10}
```

```
memblist = Table[Random[Integer, {1, 15}], {5}]
```

```
{3, 13, 4, 9, 3}
```

These 2 lists will be our "power" test lists, which we will use to test the efficiency of our implementations:

```
powertestlst = Range[4000];
powermemblist = Range[1000, 3000];
```

6.5.3 Procedural solutions

We will start with a couple of straightforward procedural attempts to solve our problem.

6.5.3.1 The absolutely worst way to do it

Below is the absolutely worst possible solution for this problem.

```
Clear[mapOnMembersProcBad];
mapOnMembersProcBad[f_, x_List, y_List] :=
  Module[{i, j, result = {}},
   For[i = 1, i ≤ Length[x], i ++,
    For[j = 1, j ≤ Length[y], j ++,
     If[SameQ[x[[i]], y[[j]]],
       AppendTo[result, f[x[[i]]]]; Break[]];
    ];
    If[j == Length[y] +1, AppendTo[result, x[[i]]]]
   ];
   Return[result]];
```

Check :

```
mapOnMembersProcBad[f, testlst , memblist]
```

```
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
```

```
mapOnMembersProcBad[f, powertestlst , powermemblist]; // Timing
```

```
{60.859 Second, Null}
```

- 6.5.3.2 A marginally better procedural way

Here we will pre-allocate the list and get rid of Append operators, in the hope that this will help

```
Clear[mapOnMembersProc];
mapOnMembersProc[f_, x_List, y_List] := Module[{i, j, copy = x},
   For[i = 1, i ≤ Length[x], i ++,
    For[j = 1, j ≤ Length[y], j ++,
     If[SameQ[x[[i]], y[[j]]], copy[[i]] = f[x[[i]]]; Break[];]]];
   Return[copy]];
```

Check

```
mapOnMembersProc[f, testlst , memblist]
```

```
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
```

```
mapOnMembersProc[f, powertestlst , powermemblist]; // Timing
```

```
{59.812 Second, Null}
```

There is almost no gain here, since the cost of Append turns out to be negligible w.r.t. cost of sweeping through a double-loop.

6.5.3.3 A real improvement: version with a binary search

Of course, it would not be fair to end the procedural part with only the above implementations, since everybody knows that there are better ways to test for membership. In particular, we could use binary search instead of a linear one. It may seem that there could be a problem here since first, both lists < x > and < y > have to be completely generic (this is our goal), second, the list < y > has to be sorted, and third, the binary search has to use the same comparison function as the one used for sorting the list. Fortunately for us, the built - in Sort is a generic function, and the built - in generic OrderedQ predicate gives exactly the same results as the comparison function used internally by Sort (when no explicit comparison function is applied to it). So, here is a generic version of binary search (this is a modified version of the code from the book of Roman Maeder):

```
Clear[bsearch];
bsearch[set_List, elem_] :=
  Module[{n0 = 1, n1 = Length[set], m, temp},
   While[n0 ≤ n1,
    m = Floor[(n0 + n1) / 2];
    If[SameQ[temp = set[[m]] , elem], Return[m]];
    If[OrderedQ[{temp, elem}], n0 = m + 1, n1 = m - 1]];
   Return[0]];
```

It will return a position of the (first entry of the) element if it is found, and 0 if not. Then, here is our modified mapping function:

```
Clear[mapOnMembersProcBS];
mapOnMembersProcBS[f_, x_List, y_List] :=
  Module[{i, copy = x, sortedy = Sort[y]},
   For[i = 1, i ≤ Length[x], i ++,
    If[bsearch[sortedy, x[[i]]] ≠ 0, copy[[i]] = f[x[[i]]];]];
   Return[copy]];
```

Check:

```
mapOnMembersProcBS[f, testlst , memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
mapOnMembersProcBS[f, powertestlst , powermemblist]; // Timing
{1.362, Null}
```

The timing is quite decent, and this is perhaps where users coming from the procedural background would stop.

6.5.3.4 Using MemberQ in a procedural version

It is interesting to see what we get if we use instead of binary search the built-in MemberQ predicate:

```
Clear[mapOnMembersProcBetter];
mapOnMembersProcBetter[f_, x_List, y_List] :=
  Module[{i, copy = x},
   For[i = 1, i ≤ Length[x], i ++,
    If[MemberQ[y, x[[i]]], copy[[i]] = f[x[[i]]]]];
   Return[copy]];
```

Check

```
mapOnMembersProcBetter[f, testlst , memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
mapOnMembersProcBetter[f, powertestlst , powermemblist]; // Timing
{2.023, Null}
```

We can appreciate the power of the built-in function - we get a 30 times improvement straight away (given that internally MemberQ is bound to perform a linear search, being a general-purpose operation)! This is not so bad, but first, it still has a linear complexity in the size of the <y> list (rather than logarithmic for binary search, and the cost of a single Sort command is negligible both because it is only done once and because it is highly optimized). Second, it turns out that we can do a lot better.

▪ 6.5.4 Functional implementations

▪ 6.5.4.1 Using Map and MemberQ

We now turn to functional implementations. The simplest that comes to mind is to use the standard Map for mapping, If to choose which function to Map, and MemberQ to test membership in the second list:

```
Clear[mapOnMembersFun1];
mapOnMembersFun1[f_, x_List, y_List] :=
  Map[If[MemberQ[y, #], f[#], #] &, x];
```

Check:

```
mapOnMembersFun1[f, testlst , memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
mapOnMembersFun1[f, powertestlst , powermemblist]; // Timing
{2.012, Null}
```

We see that this does not make a difference w.r.t. the previous improved procedural realization, which tells us that the most expensive operation is now MemberQ.

■ 6.5.4.2 Using Map and Alternatives

As another attempt of the similar kind, we can replace MemberQ by explicitly constructed from the second list large alternative pattern, just like in example of section 4.3.3.4:

```
Clear[mapOnMembersFun2];
mapOnMembersFun2[f_, x_List, y_List] :=
  With[{alts = Alternatives @@ y},
   Map[If[MatchQ[#, alts], f[#], #] &, x]];
```

Check:

```
mapOnMembersFun2[f, testlst , memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
mapOnMembersFun2[f, powertestlst , powermemblist]; // Timing
{3.185, Null}
```

We get about 50-70% of improvement this way, essentially because explicit pattern is more precise and completely syntactic (this refers to the version 5.x. For version 6, we get for some reason the opposite effect). Note that the same replacement could also be done in a procedural version with the same effect.

■ 6.5.4.3 A functional version with a binary search

We can use binary search also in a functional implementation :

```
Clear[mapOnMembersFunBS];
mapOnMembersFunBS[f_, x_List, y_List] :=
  With[{sortedy = Sort[y]},
   Map[If[bsearch[sortedy, #] ≠ 0, f[#], #] &, x]];
```

Check:

```
mapOnMembersFunBS[f, testlst , memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
mapOnMembersFunBS[f, powertestlst , powermemblist]; // Timing
{1.352, Null}
```

The performance is the same as in the procedural one, since however fast the binary search is, it is still the most expensive operation and the overhead induced by list indexing in explicit loop is negligible.

There is nothing more we can do if we don’t get some new ideas. The bad thing in our present implementations is that the MemberQ operation is repeated afresh for every element in the first list. This leads to a complexity which is proportional to the product of lengths of the two lists. This is because, however good and optimized the MemberQ operation is, it is a general purpose function and thus it is bound to have no better than a linear complexity in the length of the second list. The fact that it is a built-in does not mean that it can do magic.

have no better than a linear complexity in the length of the second list. The fact that it is a built-in does not mean that it can do magic.

6.5.4.4 Using Intersection, Position, MemberQ and MapAt

It would be nice if we could test the membership of all elements of the first list in the second list at once. Once we start thinking in this direction, the first thing that comes to mind is to locate the set of all elements in the first list that are members also of the second list. This is very easy - we just need to use the built-in Intersection command. Once this set of elements is located, we then have to find their positions in the first list (this can be accomplished with Position), and then use more precise version of Map - MapAt, to map the function on these positions only. This is the implementation:

```
Clear[mapOnMembersFun3];
mapOnMembersFun3[f_, x_List, y_List] :=
  MapAt[f, x, Position[x, z_ /; MemberQ[Intersection[x, y], z]]];
```

Check:

```
mapOnMembersFun3[f, testlst, memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
```

If you don’t see a problem in this code, think again. But the best judge is the performance test, of course:

```
mapOnMembersFun3[f, powertestlst, powermemblist]; // Timing
{5.237, Null}
```

This looks like we moved a few steps back. I used this opportunity to illustrate once again one of the very common mistakes made when patterns are constructed: in the way it is coded, the Intersection command will be recomputed every time the pattern is checked against a new list element. In fact, it is surprising that it is so fast as to slow down our function less than one could expect. This is a proper way of doing this - preallocate the result:

```
Clear[mapOnMembersFun4];
mapOnMembersFun4[f_, x_List, y_List] :=
  With[{int = Intersection[x, y]},
   MapAt[f, x, Position[x, z_ /; MemberQ[int, z]]]];
```

Check now:

```
mapOnMembersFun4[f, testlst, memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
```

```
mapOnMembersFun4[f, powertestlst, powermemblist]; // Timing
{0.972, Null}
```

■ 6.5.4.5 Using Intersection, Position, Alternatives and MapAt

As we have noticed, in our setting we can win some performance by using Alternatives instead of MemberQ. We can try it here:

```
Clear[mapOnMembersFun5];
mapOnMembersFun5[f_, x_List, y_List] :=
  MapAt[f, x, Position[x, Alternatives @@ Intersection[x, y]]];
```

Check now:

```
mapOnMembersFun5[f, testlst, memblist]
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
mapOnMembersFun5[f, powertestlst, powermemblist]; // Timing
{0.761, Null}
```

Our new function is about twice more efficient than the previous one (the result depends somewhat on the version - 5.x or 6). Also, note that in this case there was no need in pre-allocating the result of the Intersection operation. Do you understand why?

■ 6.5.5 Yet faster implementations - read if you enjoy hacking

The last version of our function (or the version with a binary search, for larger lists) is where most users (including myself) will normally stop. It is relatively efficient, and in fact (although I deliberately went down this road starting from the worst procedural code for pedagogical purposes), with some experience this will perhaps be the first or second thing which comes to mind. But now I suggest to explore a little bit the "twilight zone of *Mathematica* hackery" and see what else we can do in principle.

■ 6.5.5.1 Boosting MapAt

As a first step, let us recall that the MapAt function can be quite slow if it has to map on many positions in a flat list at once (see section 5.2.3.3). We have so far no evidence of the extent to which this affected the performance of our implementations, but there is only one way to find out. While for the general case, improving MapAt from within *Mathematica* is a tough problem, improving it in the case of flat list is relatively simple. We will use the capability of the Part function to change a large number of expression elements in place (see section 3.6.6.2). Thus, what we have to do is the following: 1. extract from the element list all elements with the positions in a position list, with the use of Part. 2. Map the function f on a list of extracted elements 3. Use part to "place back" the elements with the function <f> already mapped on them. Also, we will have to create a copy of the original list and modify a copy. So, here is the code:

```
Clear[fastMapAtSimpleList];
fastMapAtSimpleList[f_, x_List, positions_List] :=
  Module[{copy = x}, copy[[positions]] = Map[f, x[[positions]]];
   Return[copy]];
```

Notice that the list of positions has to be as the one used in Part rather than in Extract, Position or built-in

MapAt. Thus, the syntax of our version of MapAt is somewhat different. Check:

```
fastMapAtSimpleList[f, Range[15],
 Flatten[Position[Range[15], _ ? OddQ]]]
```

```
{f[1], 2, f[3], 4, f[5], 6, f[7],
 8, f[9], 10, f[11], 12, f[13], 14, f[15]}
```

Ok, it works, but is it really faster than the built-in MapAt? Let us make a power test:

```
mapattestlist = Range[10 000];
posliststandard = Position[mapattestlist, _ ? OddQ];
poslist = Flatten[posliststandard];


fastMapAtSimpleList[f, mapattestlist, poslist] // Short // Timing
```

```
{0.01, {f[1], 2, <<9996>>, f[9999], 10 000}}
```

```
MapAt[f, mapattestlist, posliststandard ] // Short // Timing
```

```
{1.472, {f[1], 2, <<9996>>, f[9999], 10 000}}
```

The difference is more than a 100 times for this list size.

The final comments here are the following: first, our version of MapAt does not support the feature that when identical positions are present in the list, the mapped function is nested several times on the element with such at such a position. This can be implemented also, if needed, but will slow down the function somewhat. Second, the order in which the function is mapped on the elements here, corresponds to the original order of positions in the position list. For the built-in MapAt, however, the order is always depth-first, which in a flat list corresponds to a left-to-right order. This difference may matter if the function being mapped contains side effects. Again, if the latter behavior is needed, it can be easily implemented - for a flat list it amounts to just sorting the position list with Sort before using it. Again, this will slow down the function a bit. Also, note that if our version of MapAt is used in conjunction with Position command, this is unnecessary altogether, since Position by itself produces a list of positions which corresponds to a depth-first traversal of an expression (just because Position traverses expressions depth-first).

■ 6.5.5.2 Using the boosted version of MapAt

We can now use our new function to see whether we get any improvements:

```
Clear[mapOnMembersFun6];
mapOnMembersFun6[f_, x_List, y_List] := fastMapAtSimpleList[f, x,
   Flatten[Position[x, Alternatives @@ Intersection[x, y]]]];
```

Check now:

```
mapOnMembersFun6[f, testlst , memblist]
```

```
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
```

```
mapOnMembersFun6[f, powertestlst , powermemblist]; // Timing
{0.521, Null}
```

We see about 30% improvement for this size of the list. In fact, one may check that this number will remain mostly constant as we increase the list size, which means that the most expensive operation now is the Position.

■ 6.5.5.3 Positions of member elements - developing the memberPositions function

It looks like we have reached the full stop: Position is a built-in operation specially designed for finding many positions at once, and we have supplied a large alternative pattern which speeds-up the pattern-matching. However, we can do better. Let us think if we can write our own function that will find all the positions of the member elements. To develop such a function, let us start once again with a simple member list and test list

```
testlst
{1, 5, 3, 6, 7, 2, 4, 2, 8, 14, 12, 13, 4, 14, 2, 4, 4, 13, 9, 10}
memblist
{3, 13, 4, 9, 3}
```

As a first step, let us find an ordering permutation which is needed to sort <testlst>, using the standard Ordering command:

```
ord = Ordering[testlst]
{1, 6, 8, 15, 3, 7, 13, 16, 17, 2, 4, 5, 9, 19, 20, 11, 12, 18, 10, 14}
```

The numbers here indicate a sequence of positions, so that if we extract the elements at these positions in this order, we get a sorted list:

```
sorted = testlst[[ord]]
{1, 2, 2, 2, 3, 4, 4, 4, 4, 5, 6, 7, 8, 9, 10, 12, 13, 13, 14, 14}
```

Here, I used the capability of Part to extract many elements at once.

By using a well-known for us by now combination of Split, Length, Transpose and Map (see section 3.10.3.4), we can obtain a list of unique (distinct) elements plus a list of their frequencies (which are given just by lengths of the sublists of same elements which Split produces):

```
{distinct, freqs} = Transpose[{#[[1]], Length[#]} & /@ Split[sorted]]
{{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14},
 {1, 3, 1, 4, 1, 1, 1, 1, 1, 1, 1, 2, 2}}
```

Now we would like to know, to which intervals of positions in the original sorted list correspond the sublists of identical elements produced by Split. Since the list of frequencies is at the same time the list of lengths of these sublists, the general formulation of our sub-problem is to obtain a list of position intervals given a partition of the length of the main list into lengths of sublists. For example, for a list Range[10] and length partitioning {1,3,4,2}, we should get the following position list : {{1,1},{2,4},{5,8},{9,10}}. The way to solve this problem is to construct partial sums of the length list by using the FoldList. This will give the starting points of the intervals when we remove the last element, and the endpoints when we subtract 1 from this list and remove the first element. Then we need to Transpose the resulting two lists. So, here is the code:

and the endpoints when we subtract 1 from this list and remove the first element. Then we need to Transpose the resulting two lists. So, here is the code:

```
posints =
 Transpose[{Most[#], Rest[#] -1} &[FoldList[Plus, 1, freqs]]]
{{1, 1}, {2, 4}, {5, 5}, {6, 9}, {10, 10}, {11, 11}, {12, 12},
 {13, 13}, {14, 14}, {15, 15}, {16, 16}, {17, 18}, {19, 20}}
```

By using a pure function like this, we can avoid an introduction of an auxiliary variable to hold the result of FoldList operation. This is a generally useful trick.

What we would like to do now is to create a set of rules, relating each distinct element in a list to an interval of positions where this element (identical copies of it) is present in a sorted list. This is done in a standard way using Thread (see section 5.3.1.5).

```
rules = Thread[Rule[distinct, posints]]
{1 → {1, 1}, 2 → {2, 4}, 3 → {5, 5}, 4 → {6, 9}, 5 → {10, 10},
 6 → {11, 11}, 7 → {12, 12}, 8 → {13, 13}, 9 → {14, 14},
 10 → {15, 15}, 12 → {16, 16}, 13 → {17, 18}, 14 → {19, 20}}
```

The next and absolutely crucial step is to use Dispatch, to create a hashed version of our set of rules:

```
rules = Dispatch[rules]
Dispatch[{1 → {1, 1}, 2 → {2, 4}, 3 → {5, 5}, 4 → {6, 9}, 5 → {10, 10},
  6 → {11, 11}, 7 → {12, 12}, 8 → {13, 13}, 9 → {14, 14}, 10 → {15, 15},
  12 → {16, 16}, 13 → {17, 18}, 14 → {19, 20}}, -DispatchTables -]
```

Now we can find members of the first list which are also members of the second list, as before, by using Intersection:

```
members = Intersection[memblist, testlst]
{3, 4, 9, 13}
```

The next step is to find intervals of positions in the sorted list which correspond to these elements. We use our Dispatched rules for that:

```
ints = ReplaceAll[members, rules]
{{5, 5}, {6, 9}, {14, 14}, {17, 18}}
```

Now we will use Range with the Map[Apply,..] (@@@, see section 5.2.7.5), to generate all the positions from position intervals:

```
positions = Range @@@ ints
{{5}, {6, 7, 8, 9}, {14}, {17, 18}}
```

We will also Flatten this list, since we no longer need the internal braces:

```
positions = Flatten[positions]
{5, 6, 7, 8, 9, 14, 17, 18}
```

To get a corresponding list of positions of these elements in the original unsorted list, we recall that we have an access to an ordering permutation. All we have to do is just to extract from this permutation the elements (positions in an unsorted list) which are at the positions we have just found. This is perhaps the most logically non-trivial step in the whole procedure and may take a bit to digest. Anyway, here is the result:

```
mempositions = ord[[positions]]
{3, 7, 13, 16, 17, 19, 12, 18}
```

The final step is to Sort these positions:

```
result = Sort[mempositions]
{3, 7, 12, 13, 16, 17, 18, 19}
```

Let me display both lists again so that we can see that these positions indeed are the positions of he common members of the two lists, in the first list:

```
testlst
{1, 5, 3, 6, 7, 2, 4, 2, 8, 14, 12, 13, 4, 14, 2, 4, 4, 13, 9, 10}
memblist
{3, 13, 4, 9, 3}
```

This was a terribly long discussion (it actually took me several times less time to write this function than to describe it), but let us now condense all the steps into a single function:

```
Clear[memberPositions];
memberPositions[x_List, y_List] :=
  Module[{order, xsorted, distinct, freqs,
    rules, positionsInSorted, posintervals},
   xsorted = x[[order = Ordering[x]]];
   {distinct, freqs} =
    Transpose[{#[[1]], Length[#]} & /@ Split[xsorted]];
   posintervals = Transpose[{Most[#], Rest[#] -1} &[
      FoldList[Plus, 1, freqs]]];
   rules = Dispatch[Thread[Rule[distinct, posintervals]]];
   positionsInSorted =
    Flatten[Range @@@ ReplaceAll[Intersection[x, y], rules]];
   Return[Sort[order[[positionsInSorted]]]]];
```

Let us check again that it gives the right thing:

```
testlst
{1, 5, 3, 6, 7, 2, 4, 2, 8, 14, 12, 13, 4, 14, 2, 4, 4, 13, 9, 10}
memberPositions[testlst, memblist]
{3, 7, 12, 13, 16, 17, 18, 19}
testlst[[memberPositions[testlst, memblist]]]
{3, 4, 13, 4, 4, 4, 13, 9}
```

The function we have just developed represents some value by itself, but now we will use it in our problem.

6.5.5.4 Using memberPositions function

Now we can try to use our newly developed function. The new code will look like:

```
Clear[mapOnMembersFast];
mapOnMembersFast[f_, x_List, y_List] :=
  fastMapAtSimpleList[f, x, memberPositions[x, y]];
```

We now check it:

```
mapOnMembersFast[f, testlst, memblist]
```

```
{1, 5, f[3], 6, 7, 2, f[4], 2, 8, 14, 12,
 f[13], f[4], 14, 2, f[4], f[4], f[13], f[9], 10}
```

```
mapOnMembersFast[f, powertestlst, powermemblist]; // Timing
```

```
{0.08, Null}
```

We see that we have made a quantum improvement - our function is now 7-10 times faster than the best of our previous implementations. I have to add that a lot of it is due to our use of Dispatch. You can try removing it and you will see that the performance will greatly drop down. Notice by the way, that in this example (and for the given size of the test list), the difference in performance between the present best version and the worst procedural one in 2000 times (on my machine). Of course, this number is not really a constant and will increase for large lists and decrease for smaller one. The real story is that we obtained a solution with a different computational complexity. This means that we will benefit from this solution even more if the intersection of the two large lists is large:

```
powertestlst1 = Range[10 000];
powermemblist1 = Range[5000, 20 000];
```

```
mapOnMembersFast[f, powertestlst1, powermemblist1] // Short // Timing
mapOnMembersFun6[f, powertestlst1, powermemblist1] // Short // Timing
mapOnMembersFun5[f, powertestlst1, powermemblist1] // Short // Timing
mapOnMembersFun4[f, powertestlst1, powermemblist1] // Short // Timing
mapOnMembersFun2[f, powertestlst1, powermemblist1] // Short // Timing
mapOnMembersFun1[f, powertestlst1, powermemblist1] // Short // Timing
```

```
{0.2, {1, 2, <<9996>>, f[9999], f[10 000]}}

{2.734, {1, 2, <<9996>>, f[9999], f[10 000]}}

{5.318, {1, 2, <<9996>>, f[9999], f[10 000]}}

{6.659, {1, 2, <<9996>>, f[9999], f[10 000]}}

{71.623, {1, 2, <<9996>>, f[9999], f[10 000]}}

{38.044, {1, 2, <<9996>>, f[9999], f[10 000]}}
```

This is a result of timings for version 6. In version 5.2, the fastest function is even much faster (about twice), and the difference between slow and fast functions is even more dramatic

6.5.5.5 Another application of <memberPositions>: the <unsortedIntersection> function

The memberPosition function that we have developed, is useful by itself, and can be used for other purposes as well. For instance, in the section 4.3.3.4 we have considered an unsorted Intersection function (that is, the function that returns all elements of the first list that are also members of the second one, but in the same order in which they are present in the first list, and also it does not remove the duplicate elements from the result). Our previous implementation was based on Cases and either MemberQ or Alternatives, very similarly to our functional versions of <mapOnMembers>. And, like them, it also can be sped up by using memberPositions. Here is the code:

```
Clear[unsortedIntersectionFast];
unsortedIntersectionFast[x_List, y_List] :=
  x[[memberPositions[x, y]]];
```

Basically, all it has to do it to extract the elements from the first list, given the list of positions computed by <memberPositions>. Check:

```
unsortedIntersectionFast[Range[20, 1, -1], Range[10]]
{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

We could revisit our previous implementation and convince ourselves that this one is much superior in terms of efficiency.

6.5.6 Conclusions

As I mention in the Preface, in my opinion *Mathematica* programming is divided in 3 layers in terms of efficiency: scripting, intermediate and system layers. This section is a good illustration of this statement: solutions in the first part (procedural) are "scripting" in the sense that they are completely straightforward. Also, they are slow, and can practically be used only on very small lists.The unfortunate thing is that they are also the ones which are most likely to first come to mind for people coming from the procedural background. Solutions in the second part (functional, using Intersection, MemberQ or Alternatives) represent an "intermediate" level and are generally not bad, and also easy to write for anybody with some *Mathematica* programming experience. For many purposes they can be good enough.

The final solution which is based on boosted MapAt and memberPositions functions, represents a "system" level (if there are further ways to speed up this code within *Mathematica*, I am not aware of them), and can perhaps be packaged to make an extension of *Mathematica* language. While not as fast as the analogous built-in probably would be, it should be fast enough for most purposes for which *Mathematica* is generally acceptable in terms of performance. In terms of development speed, it takes somewhat longer to get the "system level" function done, mainly to figure out the idea of the implementation. Perhaps, with experience it takes about couple of minutes to make any of the functional versions work (in fact, faster than the procedural ones since the code is shorter) and about half an hour to get the structural version done (well, at least in my experience. Perhaps it can be done much faster still). But given the level of generality of the problem in question, I think this is acceptable.

Returning to the problem in question, the final comment is that in cases when the mapped function is very computationally intensive (so that the computation of this function becomes the most expensive operation), the difference in efficiency between the various solutions we have discussed will matter less or much less. So, it makes sense to perform this kind of analysis before going say from the "intermediate" to "system" levels in one’s implementations, because it may be just not worth it for a given problem.

the difference in efficiency between the various solutions we have discussed will matter less or much less. So, it makes sense to perform this kind of analysis before going say from the "intemediate" to "system" levels in one's implementations, because it may be just not worth it for a given problem.

6.6 Summary

In this chapter we went through a lot of code of type common to see in real *Mathematica* programs. My goal was to illustrate several things, such as the typically large number of ways to solve a given problem in *Mathematica*, and the relative efficiency of these solutions. The rule of thumb is that the procedural programming style is more likely to produce inefficient solutions, the functional programming style is usually more efficient (when applied with some thought), and the programming based on the structural operations is typically the most efficient style, which sometimes can improve performance by an order of magnitude or lead to a solution of different computations complexity altogether.

Also, one could naively expect that the code size of the best solution should be larger than that of the worst one (because the former has to be more sophisticated than the latter). This is generally not so. The best solution is the one which achieves the closest correspondence between the structure of the problem and the efficient structures and operations built in *Mathematica*. But due to a very high level of the language, this does not necessarily imply longer code, and often in fact implies just the opposite.

We have also seen that many of our best solutions resulted from the complementary use of several different programming styles, such as structural operations, functional programming, rule-based programming (especially in combination with hashing through Dispatch). This is perhaps where the most interesting and efficient solutions in *Mathematica* should be, since this possibility to mix different programming styles is one of the unique features of *Mathematica* system.

Appendices

■ Appendix A: what is so special about *Mathematica* (a personal evaluation)

Mathematica is distant from many other programming languages in many ways. One that seems most important to me is that it is not a minimal language, in the sense I will explain. This leads to lots of ways of how any given problem can be solved. *Mathematica* language supports all major programming styles - procedural, rule-based, functional and object-oriented (OO is not directly supported but can be implemented within *Mathematica* [3,4,1]) . This richness of the language is a great strength since it allows to choose the programming style which is best suited for a particular problem - some people say that *Mathematica* language is "problem-oriented".

It allows one to program and research at the same time, using in the research, in principle, all the power of modern mathematics. The great advantage here is that it is very easy to switch the thinking mode from programming to research and back, or do necessary (non-trivial) mathematical (or statistical, etc) checks quickly without resorting to special libraries and interrupting the main programming workflow.

The problem (pun unintended) is however that all the different solutions possible for a given problem, are inequivalent, primarily in terms of efficiency, but also in terms of the code readability, ease of maintenance and debugging. These issues will probably be of no concern for a pure scientist who just needs to plot a graph or two, simplify some formula etc. But this will certainly be a concern for people from the software development community who may consider to use *Mathematica* as a tool for rapid prototyping, for which, in my opinion, it has major advantages for complex software.

When I program in C, and say solve some problem in two different ways, it is not very likely that the performance of the two implementations will be different more than a factor of 2 (unless I do something stupid, or when the difference in solutions will be actually in algorithms of different complexity). In *Mathematica* however, it is quite easy to get say 5 or 10 different solutions where the performance of the most and least efficient may differ in several orders of magnitude or have different computational complexity altogether. Of course, the reason is that, programming in *Mathematica*, we "sit" on top of a lot of internal algorithms used to implement given built-in functions that we are using. But from the user viewpoint, these performance differences are often completely unobvious, until one gets a better understanding of how the system works.

Imagine now that we are building a system which has 4 stages of information processing, each one taking as an input a result from the previous one. And then, on each stage we produce 2 different solutions which differ in performance 10 times. At the end, we get one system working 10000 faster than the other. In practice, this means that the "slow" system in most cases will be completely useless, given that there anyway exists an overhead due to the symbolic nature of *Mathematica* and a very high level of its language. If one wants to build something serious and interesting in *Mathematica*, one has to learn techniques to program in it efficiently, or at the very least, be aware of certain performance pitfalls associated with each programming style.

At the same time, the great advantages that *Mathematica* brings are the ease and speed of writing and debugging the code, the extremely small code size, and the ability to stay on quite a high level of abstraction throughout the process of solving the problem, without going into unnecessary low-level details which hide the essence of the problem. This allows a single person to manage substantial projects. *Mathematica* is a great "thinking laboratory". Due to its highly interactive nature, it is also a great tool to design and

At the same time, the great advantages that *Mathematica* brings are the ease and speed of writing and debugging the code, the extremely small code size, and the ability to stay on quite a high level of abstraction throughout the process of solving the problem, without going into unnecessary low-level details which hide the essence of the problem. This allows a single person to manage substantial projects. *Mathematica* is a great "thinking laboratory". Due to its highly interactive nature, it is also a great tool to design and analyze algorithms. For me personally, this overweights the eventual complications arising from the above performance issues, especially because once you understand the system, you rarely get an unexpected behavior or nasty performance surprises.

Let us forget for a while about the most well-known goal of *Mathematica* - to carry out mathematical transformations and solve various mathematical problems (symbolically or numerically or both), and think of it as a programming environment. The main questions then are: what are the main ingredients of this environment, how are they different from their analogs in other languages, what sort of problems can be solved better or easier, and which programming paradigms and ways of thinking are encouraged in *Mathematica*.

If I was asked to describe *Mathematica* in one sentence, I would say that it represents a functional programming language built on top of a rule-based engine, which operates on general symbolic trees (or, directed acyclic graphs if you wish).

Mathematica mainly consists of the following blocks:

1. Powerful rule-based engine with pattern-matching and evaluator built around the general *Mathematica* expressions - we can think of this as a programming environment defined on and optimized to work with general symbolic trees.

2. Global rule base which allows the user to both define functions as global rules and make them interact with the pre-built system rules in a non-trivial way. The former is a necessary ingredient for programming. The latter can be used effectively in, for instance, carrying out mathematical transformations and simplifications, since the system already knows many identities and properties of functions and other mathematical objects. However, it can be used in many more situations, basically every time when we want to define new objects by new rules. Systems of rules are way more flexible than say classes in OO paradigm, since they basically define grammars of small languages, and are not rigidly tied to specific data structures.

Function calls are then internally just a special instance of application of global rules. They are made efficient by built-in hash tables used for global rules (among other things). Type checking (when needed) is made almost trivial by the pattern-matcher. The functions can be "overloaded" in much more general way than in more traditional OO languages.

3. Highly optimized and efficient structural operations on lists and arrays (Flatten,Transpose, Partition, all numerical built-in functions, comparison operations, etc), which are similar to those in the APL language.

4. Support of the functional programming paradigm by both the possibility of defining pure (anonymous) functions and by efficient built-in higher-order functions such as Apply, Map, Fold, etc. Due to the uniform representation of everything as *Mathematica* expression, these higher-order functions (and thus the FP programming style) apply to general *Mathematica* expressions rather than just lists. This is a very powerful capability.

The availability of the rule-based approach means basically that one can easily create a language describing any new object one wants, be it either a more formal language with a grammar or just a collection of some objects and relations between them. What is important is that this can be completely syntax-based. By adding new rules to some built-in functions ("overloading") one can make this new language immediately interact with rules that exist in *Mathematica* kernel and thus take advantage of those.

The rule-based approach also means that the language is very unrestrictive (or should I say powerful) - it puts virtually no bounds on what types of manipulations can be done in principle. As some extreme examples, one can define functions that produce other functions, functions that change their own definitions at run-time (for example, we may program a function that destroys itself after it is done with the work, and even produce such "disposable" functions at run-time by other functions), functions that manipulate the definitions of other functions at run-time, and many more seemingly weird possibilities. Techniques like dynamic programming, caching and memoization, lexical closures etc are a common practice in *Mathematica* programming and require little effort from the programmer. Also, if one feels that for a particular problem a more "rigid" or restrictive framework (such as object orientation) is needed, it can be implemented within *Mathematica*.

The availability and effectiveness of functional programming style allows to both make the code more concise and create data structures on the fly (since in this approach any complex data structure is represented by a possibly nested list). If however one wants to shift the accents more towards data structures, this is also possible and easy thanks to the syntax-based pattern-matching and rule substitution. And because in *Mathematica* functional programming can be performed on general *Mathematica* expressions (more general than lists - this is made non-trivial by pattern-matching), one can also combine the two programming styles to shift the relative roles of functions and data structures as to feel most comfortable. It is typical in *Mathematica* programming to use functional programming in the more exploratory stage and then create more rigid data types and structures after the design has shaped.

The large number of built-in functions has both advantages and disadvantages. To list just a few advantages: you get a huge collection of (often very sophisticated) algorithms already implemented, tested etc., packaged in built-ins. Extended Help system and error messages allow to very quickly learn new functionality, write and debug programs. However, while the capabilities of *Mathematica* such as pattern-matching and rules substitution are great, they are also expensive in terms of performance. As a result, many operations would be too slow if implemented directly in *Mathematica* language. Therefore, they are implemented in a lower-level language such as C, and packed into the *Mathematica* kernel. This solves the problem, but often makes the performance hard to understand (especially for inexperienced users), since the performance of user-defined and built-in functions can be dramatically different.

All is not lost however. The general principles on which *Mathematica* is built give the language overall consistency. This, plus a large number of quite generic and efficient built-in higher-order functions (that is, functions that manipulate other functions) allow for efficient general *Mathematica* programming techniques. These techniques are not too difficult to learn, and in some sense they split the entire *Mathematica* language into a "scripting" (quick to write, but often slow to execute), "intermediate" (a bit more thinking but faster code), and "system" (less intuitive thinking, but yet much faster code) language layers (please bear in mind that this classification is my own and based on my personal experience, rather than a widely accepted one).

The part of the difficulty of learning *Mathematica* programming is that there is no good formal distinction between these layers. Typically, the first is characterized by heavy use of the procedural (or otherwise straightforward) code, the second corresponds to use of functional programming and the third by heavy use of optimized structural operations, but this is not an absolute criteria. One and the same operation can play a "scripting" role in one context and "system" role in another.

For many problems (especially purely scientific), "scripting" layer is sufficient. This layer consists mainly in using built-in commands or gluing them with a typically procedural code. A big part of the bad reputation that *Mathematica* used to have for its "slow performance" is related to the fact that most people are only aware of this language layer, because it corresponds most directly to their programming experience in other (procedural) languages.

The other two layers serve several purposes, such as improving speed and quality of code design, generally improving performance, and removing certain performance bottlenecks within *Mathematica*, without resorting to external code (although this is also possible through connecting technologies such as *MathLink* or *J/Link).* Also, and perhaps even more importantly, they provide a programmer with new ways of thinking about the problems. Less important for some scientific applications, these layers are much more important for software development and prototype design.

From the pragmatic point of view, the proper use of each of the above capabilities individually, and the ability to choose the programming paradigm that best fits a given problem, can greatly improve one's uses of *Mathematica* (both in terms of speed of writing and debugging the program, and speed of the code execution).

It probably does not make sense to master *Mathematica* on this level for someone who needs it just occasionally, to compute an integral or two or plot a graph or two. However, for a person who needs to routinely perform lots of non-trivial checks and experiments (typical for computer modelling/simulations or rapid prototyping), this level of use of *Mathematica* will be very valuable. The end result of learning these techniques will be twofold: great reduction of time (both human and computer) and code size for most problems, and the ability to push *Mathematica* a lot further in solving hard or computationally-intensive problems, before switching to more efficient specialized software or programming language.

From the programmer's point of view, the speed of writing and debugging the code combined with its typically small size allows a single person to manage quite large projects. The mechanism of packages provides a support for larger scale programming. In addition, by combining the above functionality in non-trivial ways, one can develop different and possibly novel ways of both programming and thinking about problems. The underlying rule-based nature of *Mathematica* makes it possible to remove many restrictions on what can be done in principle, typical for more traditional languages. The price to pay is often efficiency issues. Getting familiar with *Mathematica* on a deeper level can help deal with them in many cases.

For non-trivial and/or computationally demanding problems, containing many steps, it is rather dangerous in my view to take the "receipe" approach and search in the Help etc for similar problems solved (this certainly helps a great deal, but you have to understand the code). Even if the solution you find is optimal for some other problem, there are many subtleties which may turn your even slightly modified code wrong or inefficient. Learning these subtleties by trial and error may be faster for every given case, but does not pay off at the end, if one has to frequently use *Mathematica*. On the other hand, learning a coherent picture of *Mathematica* programming will ensure that you always pick the right idiom for the problem. Also, all the mentioned subtleties are then naturally understood within this framework, since on a deeper level they simply reflect the way the system works.

Appendix B Some of my favorite books on *Mathematica* - brief reviews

Here I would like to review some selected books. The choice of books to review is very personal. It is very likely that there are some very good books missing here which I simply didn't have a chance to look at. Also, I left out all books which are field-specific - the books below are all focused on *Mathematica* programming proper (although many of them contain lots of examples of how *Mathematica* can be applied to solve problems in various fields). Also, one should not consider the reviews below as some advice to prefer one book to another. All these books equally belong to the shelf of the *Mathematica* programmer, and each has its unique strength and value, which I tried to briefly describe below.

The *Mathematica* Book, by Stephen Wolfram

Written by the creator of *Mathematica*, this is the most authoritative and detailed account on *Mathematica* in all its aspects. The book is written in a very clear way, and contains lots of examples. Essential for understanding the *Mathematica* principles and language. Everyone serious about *Mathematica* programming has to read it at least once. The limitation is that example problems are mostly limited to those which can be solved with a very few lines of code (in *Mathematica*, this does not mean that they are always trivial). Will not teach directly some specifics of writing larger programs.

An introduction to programming with *Mathematica*, by Paul Wellin, Richard Gaylord and Samuel Kamin

A great introductory textbook on *Mathematica* programming - probably the best in its class. Best place to start learning *Mathematica* programming if you are a complete beginner. Even if you are not, you are still likely to learn new things from it. Has a lot of overlap with the present tutorial. Covers wider range of topics, in particular notebook, frontend and graphics programming. Contains lots of quite useful exercises. Authors do not particularly emphasize efficiency (at least, not to a degree I tried to do it in this text), but are pretty careful not to include inefficient implementations without saying it. Material is organized somewhat differently though, more according to different programming styles. The authors focus more on showing how certain problems from various branches of science can be solved in *Mathematica*, while I focus more on the language features.

Programming in *Mathematica,* by Roman Maeder

A classic text on *Mathematica* programming from one of the original designers of *Mathematica*. Very elegantly written, contains lots of useful examples, particularly of larger *Mathematica* programs. Its goal is not to be complete with all the small details, but to be essential. Is a harder read than the previous book, but is readily recommended as a second one. The more one already knows about *Mathematica* programming, the more one appreciates this book. Great reference for writing packages. Contains certain discussions which are hard to find elsewhere.

- Power programming in *Mathematica* - the Kernel, by David Wagner

Probably the best book on advanced *Mathematica* programming, devoted entirely to *Mathematica* language. Written by a computer scientist, and from a computer science viewpoint. This book made a real difference in my *Mathematica* education and opened my eyes on many things. Those familiar with it will undoubtedly see its influence on the present tutorial. Explains many subtleties, tricks and techniques, and also explains why, not just how. Emphasizes performance and teaches performance-tuning techniques. This book can help make a quantum jump in one's abilities as a *Mathematica* programmer. Readily recommended to anyone willing to create non-trivial applications in *Mathematica*. While it covers version 3 only, this matters very little since almost everything it discusses remains true in newer versions. Unfortunately, went out of print.

- Mastering *Mathematica*, by John Gray

Many non-trivial applications of *Mathematica* programming in this book. A big part of it is also devoted to teach *Mathematica* programming, and is absolutely worth reading. Contains several large applications such as object-oriented graph theory in *Mathematica*. One of the very few places (apart from Maeder's books) where the object-oriented extension to *Mathematica*, developed by R.Maeder, is explained in detail and illustrated with non-trivial examples. A slight downside is that sometimes inefficient solutions are presented without warnings, but this is a minor flaw - you anyway will have to develop your own instincts for this if you enter serious *Mathematica* programming.

- *Mathematica* guidebooks, by Michael Trott

Written the *Mathematica* insider and expert, these amazing 4 volumes represent by far the most advanced treatment of *Mathematica* programming in application to numerous problems of contemporary science that I am aware of. Contains an overwhelming collection of non-trivial examples, and more small details about *Mathematica* that you will ever want to know. These books show what is possible when one is armed with the fullest power of *Mathematica.* Essential for advanced users.

Appendix C Performance of some built-in functions in certain important special cases

ReplacePart: efficiency pitfall for massive changes

ReplacePart may be very slow when replacing many elements with symbolic expressions at the same time (by many I mean several thousands or more). There is a way to cure it in some cases, which we will describe later in part II. For now, consider an example: we will change every second element of the large list with 0:

```
ReplacePart[Range[15 000], 0, List /@ Range[1, 15 000, 2]]; // myTiming
0.00234
```

```
Clear[a];
ReplacePart[Range[15 000], a, List /@ Range[1, 15 000, 2]]; //
 myTiming
2.422
```

The difference in speed of replacing a number (0) and a symbol (a) is 1000 (!) times in this example (it increases even more once we increase the list size and the number of replaced elements). I don't want to go into details here, but the reason has to do with the *Mathematica* packed arrays technology. For now, just avoid such operations in the programs you are writing, if possible.

```
Clear[testlist, complextestlist];
```

Insert: efficiency pitfall for large number of insertions

Let me mention, that just as with ReplacePart, Insert can be quite slow if you ask it to insert many symbolic expressions at once (by many I again mean at least few thousands). Once again, this can be cured, and we will consider the ways to do it in later chapters. For now let me just illustrate the difference:

First, a small example: we insert zero in every third position in a small list:

```
Insert[Range[10], 0, Map[List, Range[1, 10, 2]]]
{0, 1, 2, 0, 3, 4, 0, 5, 6, 0, 7, 8, 0, 9, 10}
```

This is how long it takes to insert zero in every third position in a list of first 10000 natural numbers:

```
Clear[a, plist];
plist = List /@ Range[1, 10 000, 2];
Insert[Range[10 000], 0, plist]; // myTiming
0.001016
```

And here we insert a symbol <a>:

```
Insert[Range[10 000], a, List /@ Range[1, 10 000, 2]]; // Timing
{1.328 Second, Null}
```

The difference is more than a 1000 (!) times for this list, which is even not too large. This is also related to packed arrays. For now, once again, it is better to avoid such massive symbolic insertions. For smaller lists,and also for lists of numbers, Insert works quite fast and does not have this behavior.

At the same time, the Delete function is fast regardless of the content of the list - whether it is symbolic or numeric.

- ***Union***, ***Intersection*** and ***Complement***: performance with a user-defined SameTest option

As we mentioned in the text, the operations Union, Intersection and Complement with the SameTest option may perform slower or much slower than without it. We will only illustrate this for Union, but the same is true also for the other two.

Consider some test list

```
testlist = Table[Random[Integer, {1, 10}], {15}]
```

For example, we may consider elements the same if they are either equal or differ by 2:

```
Union[testlist, SameTest → (SameQ[#1, #2] || Abs[#1 - #2] == 2 &)]
{1, 2, 5, 6, 10}
```

The topics of rules, options and pure functions, which syntax we just used, may have not been covered yet, in which case ignore the syntax details for now.

The problem is the following. The Union operation based on the default sorting function is very fast, but it may become a lot slower with a user-defined SameTest option. It shares this property with the Sort function (to be discussed next), essentially due to the way that Union operation is organized: it first sorts the list and then the same elements will always be adjacent to each other and thus easier to eliminate. To illustrate this point, consider a larger list:

```
(testlist = Table[Random[Integer, {1, 1000}], {1500}]) // Short
{735, 280, 715, <<1494>>, 415, 650, 516}
Union[testlist] // myTiming
0.000329
Union[testlist, SameTest → (SameQ[#1, #2] || Abs[#1 - #2] == 2 &)]; //
 myTiming
0.531
```

We see that it is more then a 1000(!) times slower with this non-trivial "sameness" function, for this size of the list.

This illustrates several things. First of all, if we think of it, the specific problem and the notion of "sameness" as formulated above is ill-posed, because depending on the order in which the Union operation is performed, we will get different results. For example, consider a list {2,4,6}: if <4> is eliminated first, we get {2,6}, but if <6> is eliminated first, we get just {2}. Essentially the problem here is that our notion of sameness is not transitive. Perhaps a more meaningful formulation in this case would be to locate all chains of numbers with each one different by 2 from the next one in the chain. In any case, one has to make sure that the problem is well-formulated, and the sameness function better be transitive.

However, ignoring this issue for the moment, the sameness function above illustrates another point well: it is a stronger requirement to provide a sorting function (which we did not do, and also the syntax of Union does not allow us to), than to provide the sameness function, because in the former case we have to define not just the notion of same, but also notions of greater and smaller (actually, for sorting purposes, the notion of same is less important than the latter two). However it is exactly the existence of the sorting function (criteria) which allows to map our set (list) to say natural numbers and thus reduces the computation time from quadratic in the length of the list (that is, if we just compare all elements pairwise), to log-linear. And just the fact that the built-in function Union takes the sameness function does not mean that it translates it efficiently into a sorting function - this is not always possible at all, and in any case is a (non-trivial in general) programmer's task. Thus, we should not expect miracles, but rather should reformulate the problem such that the proper sorting function is available (if possible, of course).

In fact, it is even better if we can reformulate a problem such that instead of the sorting function applied to elements of our list, we can use some key function which computes a key (say, integer number) for each element in the list, so that the majority of subsequent operations are performed with keys rather than the original elements. In *Mathematica* such approach often gives a large speed advantage since many operations are much faster with numbers than with arbitrary symbolic expressions. In cases when such key function is available, there are several techniques which can be efficiently used to replace Union. We will exploit this technique many times later, but for now let us just consider another problem as an illustration of these statements.

We will use the same large list as above, but define two numbers the same when they have the same remainder of division by 60. Here is (without explanation) the better code to eliminate the duplicate elements:

```
Reap[Sow[#, Mod[#, 60]] & /@ Sort[testlist], _, First[#2] &][[2]] //
 Short
```

```
{1, 2, 3, 4, 6, <<50>>, 161, 166, 193, 196, 239}
```

This is how long it takes to do so:

```
Reap[Sow[#, Mod[#, 60]] & /@ Sort[testlist], _, First[#2] &][[2]] //
 myTiming
```

```
0.00578
```

For versions of *Mathematica* earlier than 5.0, where Reap-Sow operations were not yet available, one can use the following code (slight extension of the technique used by Carl Woll), which takes about twice longer to execute than the one above (once again, we provide it here for illustration and timing comparison, so please ignore the code for now - we will revisit it later):

```
Module[{md, g},
  g[x_] := False;
  Map[If[g[md = Mod[#, 60]], Unevaluated[Sequence[]],
     (g[Evaluate[md]] = True); #] &, Sort[testlist]]] // Short
```

```
{1, 2, 3, 4, 6, <<50>>, 161, 166, 193, 196, 239}
```

And this is the result of Union with the SameTest option and its timing:

```
Union[testlist, SameTest → (Mod[#1, 60] == Mod[#2, 60] &)] // Short //
 Timing
{0.125 Second, {1, 2, 3, 4, 6, <<50>>, 161, 166, 193, 196, 239}}
```

We get about 25 times speed-up with Reap-Sow method and about 10 times with the Woll's technique, with respect to the one using Union, for this size of the list. If we make a list larger, the difference will be even more dramatic.

To conclude this rather long discussion, there can be a huge difference in performance of Union depending on whether it is used in its "pure" form or some "sameness" function is provided. In the latter case, and if the list is any large, it is advisable to first analyze the problem, because there could be superior alternatives. Also, this behavior is not entirely the fault of Union, but partly reflects the fact that there is no general efficient solution for eliminating same elements from the list if all we have is just the sameness function, but not a comparison function.

```
Clear[testlist];
```

- ***Sort*** : performance with a user-defined comparison function

There is an issue with the use of sorting functions for sorting large lists which I would like to mention here. The problem is that for large lists, sorting them with a custom (user-defined) comparison function may be much slower than when the default comparison function is used. The reasons have to do mostly with the way *Mathematica* is built, particularly with the fact that the larger number of consecutive operations we can "push" into the kernel (so that it does them without interruption or dialog with the higher-level symbolic constructs that we operate with in our *Mathematica* session), the more efficient code we will get. The main rule when working with lists, which we stated before, of not breaking them into pieces, is just another manifestation of this situation. Now, when we sort with the default function, kernel does all the sorting internally and just returns the result to us. But when we supply the higher-level sorting function, it has to constantly interrupt to apply this function to list elements, and it slows it down considerably. Let us consider an example to illustrate this point - we will sort a large nested list by an increasing value of the first element of the sublists:

```
(largenested = Table[Table[Random[Integer, {1, 15}],
     {Random[Integer, {3, 20}]}], {5000}]) // Short[#, 5] &
{{8, 8, 3, 3, 1, 15, 12, 5, 5, 8, 6, 13, 7, 13, 11, 7, 9, 3, 1, 4},
 {13, 10, 8, 8}, <<4997>>, {11, 10, 11, 3, 2, 4}}
```

Now we use a default sort (which will be by first element of the sublists):

```
(sorted1 = Sort[largenested]) // myTiming
0.0171
```

Here we also sort by the first element of the sublists, but this time using a built-in **OrderedQ** comparison function (which produces the same results as the one used internally by the "pure" Sort):

```
(sorted2 = Sort[largenested, OrderedQ[{#1, #2}] &]) // myTiming
0.22
```

We can check that results are the same :

```
sorted1 === sorted2
True
```

We see that in this example, the difference in speed is about 10 times (for this length of the list), but for more complicated sorting functions the difference will be even larger. Later we will see that in some cases there exists a partial solution to this efficiency problem. For small lists and when the performance is not an issue, this complication probably does not matter.

I would also like to emphasize, that I view this situation as not a result of some faults in *Mathematica* design, but rather a price to pay for having the level of generality that these functions (Sort, and also such functions as Union or Split - see below) provide - they work on lists of arbitrary *Mathematica* expressions. In cases when such level of generality is not needed, techniques are available to significantly speed-up these operations. We will discuss some of them in part II.

- ***MapAt*** : efficiency pitfall for massive mappings

One has to be aware that there is a rather strange (unexpected) performance pitfall associated with use of Position - MapAt. Consider for example mapping some function < g > on every even element in the list :

```
MapAt[g, Range[50], Position[Range[50], _ ? EvenQ]]
{1, g[2], 3, g[4], 5, g[6], 7, g[8], 9, g[10], 11, g[12], 13, g[14],
 15, g[16], 17, g[18], 19, g[20], 21, g[22], 23, g[24], 25, g[26],
 27, g[28], 29, g[30], 31, g[32], 33, g[34], 35, g[36], 37, g[38],
 39, g[40], 41, g[42], 43, g[44], 45, g[46], 47, g[48], 49, g[50]}
```

But let us consider a larger list :

```
MapAt[g, Range[1000], Position[Range[1000], _ ? EvenQ]]; // Timing
{0.02, Null}
```

And yet larger one :

```
MapAt[g, Range[25 000], Position[Range[25 000], _ ? EvenQ]]; // Timing
{24.756, Null}
```

This behavior is rather strange and this is one of the rare cases where it looks like this has been overlooked in the implementation of MapAt (however, this could be a deliberate decision or a result of some trade-off between say the speed and the reliability of the underlying algorithm used in implementing MapAt). In this case, the rule-based approach is clearly superior:

```
Range[25 000] /. x_ ? EvenQ :→ g[x]; // Timing
{0.071, Null}
```

Let me further comment on this behavior. A more detailed performance analysis shows that MapAt works quite well in cases when all branches of the symbolic tree of the expression we are mapping on do not contain too many nodes on which we have to Map. For example, it will work much better on say 10 - ary tree than on a flat list containing the same end elements. As an explicit example, consider a list of natural numbers of the length 4^7 ==2^14:

```
testexpr = Range[16 384];
pslist = Position[testexpr, _ ? EvenQ]; // Timing
```

```
{0.06, Null}
```

This is how long it takes for this flat list :

```
MapAt[g, testexpr, pslist]; // Timing
```

```
{8.592, Null}
```

Now we will partition this list into a 4 - ary tree of depth 7, containing the same numbers as its leaves :

```
(testexpr = Nest[Partition[#, 4] &, Range[16 384], 7];
   pslist = Position[testexpr, _ ? EvenQ]); // Timing
```

```
{0.28, Null}
```

Look at the timing now :

```
MapAt[g, testexpr, pslist]; // Timing
```

```
{0.09, Null}
```

We could accomplish the same task as before, by Flattening the result :

```
Flatten[MapAt[g, testexpr, pslist]]; // Timing
```

```
{0.08, Null}
```

The net speed - up in this example is 100 times, and about 30 times if we include the extra overhead of Partition command and Position operation on a nested structure.

My main conclusion here is that MapAt is optimized to work with rather "vertical" trees. In particular, it may become totally inefficient when mapping on a large list of positions, and also for mostly "flat" and "horizontal" structures characterized by branches with large number of sub - branches/leaves. In the latter situations my advice would be to switch from Mapping on certain positions to the rule-based approach, if possible. A good possibility might be to first locate the position of such large sub-branch and then use MapAt to map a rule-based substitution function on these sub-branches.

```
Clear[testexpr];
```

Appendix D Some online *Mathematica* resources

Here I briefly mention some useful *Mathematica* resources

Wolfram sites

Some of the greatest *Mathematica* resources live at several Wolfram sites:

www.wolfram.com - The main site of Wolfram Research Inc. Contains lots of information on *Mathematica* of all sorts. All other Wolfram sites link here.

MathWorld - ***http://mathworld.wolfram.com*** - An online encyclopedia of Mathematics maintained by Eric Weisstein. A great resource for various Mathematical facts, such as formulas, theorems etc. What is important is that whenever a translation of a given fact or formula to *Mathematica* is possible, it is there.

Functions site - ***http://functions.wolfram.com/*** - contains reference information on a very large number of elementary and special functions, together with visualizations and *Mathematica* notation for them.

Online documentation - ***http://reference.wolfram.com*** - The complete online documentation of all *Mathematica* features. Contains plenty of examples. The ultimate place to look for correct syntax and detailed explanations of workings of any specific function. Also has lots of small thematic tutorials.

The online repository ***http://library.wolfram.com/*** - formerly *MathSource* - contains thousands on extra materials including tutorials, application packages, technical notes and reports etc. A great resource to learn more on *Mathematica* and *Mathematica* programming. Also, may contain the application you need, already implemented completely or in part.

Other resources

Mathematica newsgroup - ***comp.soft - sys.math.mathematica*** - is the place where you can post your problem and get help from expert users or, sometimes, even developers, and also a huge repository of topics already discussed and problems solved.

Ted Ersek's Mathematica tricks - ***www.verbeia.com/mathematica/tips/Tricks.html*** - one of the best places to look for lots of unobvious details and subtleties about the workings of specific built-in commands and *Mathematica* in general. Also, a great source of examples.

The bibliography

- Books devoted mostly or entirely to *Mathematica* programming

1. John W.Gray, *Mastering Mathematica, Second Edition: Programming Methods and Applications*, Academic Press, 2 edition (1997)

2. Roman E.Maeder, *Programming in Mathematica,* Addison-Wesley Professional; 3 rd Edition. (1997)

3. Roman E.Maeder, *Computer Science with Mathematica*, Cambridge University Press (2000)

4. Roman E.Maeder, *The Mathematica Programmer,* Academic Press (1994)

5. Roman E.Maeder, *The Mathematica Programmer II*, Academic Press (1996)

6. Michael Trott, *The Mathematica Guidebook for Programming*, Springer (2004)
The Mathematica Guidebook for Graphics, Springer (2004)
The Mathematica Guidebook for Symbolics, Springer (2005)
The Mathematica Guidebook for Numerics, Springer (2005)

7. David B.Wagner, *Power Programming With Mathematica: The Kernel*, Mcgraw-Hill (1996)

8. Paul R.Wellin, Richard J.Gaylord, and Samuel N.Kamin, *An Introduction to Programming with Mathematica*, Third Edition, Cambridge University Press (2005)

9. Stephen Wolfram, *The Mathematica Book*, Wolfram Media, Fifth Edition (2003)

- Some general and introductory *Mathematica* books

10. Nancy Blachman and Colin Williams, *Mathematica : A Practical Approach*, Prentice Hall PTR; 2nd Edition (1999)

11. Heikki Ruskeepaa, *Mathematica Navigator: Mathematics, Statistics, and Graphics*, Academic Press, Second Edition (2004)

12. William T.Shaw and Jason Tigg, *Applied Mathematica : Getting Started, Getting it Done*, Addison - Wesley Professional (1993)

- *Mathematica* - unrelated references

13. Paul Graham, ANSI Common Lisp, Prentice Hall (1995)

14. http : // paulgraham.com/power.html