Think OCaml

How to Think Like a (Functional) Programmer

Version 0.1.1

www.dbooks.org

Think OCaml

How to Think Like a (Functional) Computer Scientist

Version 0.1.1

Allen Downey
Nicholas Monje

Green Tea Press

Needham, Massachusetts

www.dbooks.org

Copyright © 2008 Allen Downey.

Printing history:

April 2002: First edition of *How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

Someday: Translation to OCaml, changed title to *Think OCaml: How to Think Like a (Functional) Programmer*.

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and with no Back-Cover Texts.

The GNU Free Documentation License is available from `www.gnu.org` or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from `http://www.thinkpython.com`

Contents

1 The way of the program **1**

1.1 The OCaml programming language 1
1.2 The OCaml Toplevel 2
1.3 What is a program? 3
1.4 What is debugging? 4
1.5 Formal and natural languages 5
1.6 The first program 7
1.7 Functional, as opposed to...? 7
1.8 Debugging 8
1.9 Glossary 8
1.10 Exercises 10

2 Variables and Expressions **11**

2.1 Values and types 11
2.2 Variables 12
2.3 Variable names and keywords 13
2.4 Operators and operands 13
2.5 Expressions 14
2.6 Variables as Expressions, and References 15
2.7 Scope 15
2.8 Order of operations 16
2.9 String operations 16
2.10 Comments 17
2.11 Debugging 17
2.12 Glossary 18
2.13 Exercises 19

www.dbooks.org

3 Functions **21**

3.1 Function calls . . . 21
3.2 Math functions . . . 21
3.3 Composition . . . 22
3.4 Adding new functions . . . 22
3.5 Definitions and uses . . . 24
3.6 Flow of execution . . . 24
3.7 Parameters and arguments . . . 25
3.8 Functions are Just Renamed Expressions . . . 26
3.9 Scope and Functions . . . 26
3.10 Stack diagrams . . . 26
3.11 Currying Functions . . . 27
3.12 Why functions? . . . 28
3.13 Debugging . . . 28
3.14 Glossary . . . 28
3.15 Exercises . . . 29

4 Program Flow **31**

4.1 Modulus operator . . . 31
4.2 Boolean expressions . . . 31
4.3 Logical operators . . . 32
4.4 Conditional execution . . . 32
4.5 Chained conditionals . . . 33
4.6 Keyboard input . . . 33
4.7 Pattern Matching . . . 34
4.8 Glossary . . . 35
4.9 Exercises . . . 36

5 Recursive Functions **37**

5.1 Recursion . . . 38
5.2 Infinite recursion . . . 39
5.3 Mutually Recursive Functions . . . 40
5.4 Tail-end Recursion . . . 40
5.5 Debugging . . . 41
5.6 Glossary . . . 42
5.7 Exercises . . . 42

6 Algorithms **45**

6.1 Square roots . . . 45
6.2 Algorithms . . . 46
6.3 Debugging . . . 47
6.4 Glossary . . . 47
6.5 Exercises . . . 47

7 Strings **49**

7.1 A string is a sequence . . . 49
7.2 `String.length` . . . 50
7.3 Substrings . . . 50
7.4 String Traversal . . . 50
7.5 Searching . . . 51
7.6 String comparison . . . 52
7.7 Debugging . . . 53
7.8 Glossary . . . 53
7.9 Exercises . . . 53

8 Lists **55**

8.1 A list is a sequence . . . 55
8.2 List operations . . . 56
8.3 List iteration, mapping and folding . . . 56
8.4 Traversing a list . . . 58
8.5 List sorting . . . 58
8.6 Lists and Recursion . . . 60
8.7 Debugging . . . 61
8.8 Glossary . . . 61
8.9 Exercises . . . 61

9 Case Study: Regular Expressions **63**

9.1 File I/O . . . 63
9.2 The `Str` module . . . 65
9.3 Regular Expressions . . . 65

www.dbooks.org

9.4 Compiling Regular Expressions 67
9.5 Using Regular Expressions: Search 67
9.6 Using Regular Expressions: Replacement 68
9.7 Using Regular Expressions: Splitting 69
9.8 Debugging 69
9.9 Glossary 69

10 Putting the O in OCaml, Part 1: Imperative programming 71

10.1 References 71
10.2 Looping 72
10.3 Examples 73

11 Arrays 75

11.1 Making Arrays 75
11.2 Array Operations 76
11.3 Array Iteration, Mapping, and Folding 77
11.4 Array Sorting 77
11.5 Array Traversal 78
11.6 Glossary 78

12 Hashtables 79

12.1 Hashtable as a set of counters 81
12.2 Iteration and Hashtables 82
12.3 Folding and Hashtables 83
12.4 Reverse lookup 83
12.5 Memos 84
12.6 Long integers 86
12.7 Debugging 86
12.8 Glossary 87
12.9 Exercises 88

13 Tuples **91**

13.1 Creating and Using Tuples . . . 91
13.2 Tuple assignment . . . 91
13.3 Tuples as return values . . . 92
13.4 Lists and tuples . . . 92
13.5 Comparing tuples . . . 93
13.6 Debugging . . . 93
13.7 Glossary . . . 93

14 Records and Custom Data Structures **95**

14.1 Tuples . . . 95
14.2 Enumerated Types . . . 96
14.3 Aggregate types . . . 97
14.4 Debugging . . . 98
14.5 Glossary . . . 98
14.6 Exercises . . . 98

15 Putting the O in Ocaml Part 2: Objects and Classes **101**

15.1 Turtles . . . 101
15.2 Defining Modules and Classes . . . 103
15.3 The `Str` module . . . 104
15.4 Debugging . . . 104
15.5 Glossary . . . 104
15.6 Exercises . . . 104

16 Case study: data structure selection **105**

16.1 Word frequency analysis . . . 105
16.2 Random numbers . . . 105
16.3 Word histogram . . . 106
16.4 Most common words . . . 108
16.5 Optional parameters . . . 108
16.6 Dictionary subtraction . . . 109
16.7 Random words . . . 110

www.dbooks.org

16.8 Markov analysis 110
16.9 Data structures 112
16.10 Debugging 113
16.11 Glossary 114
16.12 Exercises 114

A Debugging **117**

A.1 Syntax errors 117
A.2 Runtime errors 119
A.3 Semantic errors 122

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program."

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 The OCaml programming language

The programming language you will learn is OCaml. OCaml is an example of a **high-level language**; other high-level languages you might have heard of are C, C++, Perl, Java and Python.

There are also **low-level languages**, sometimes referred to as "machine languages" or "assembly languages." Loosely speaking, computers can only execute programs written in low-level languages. So programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

The advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

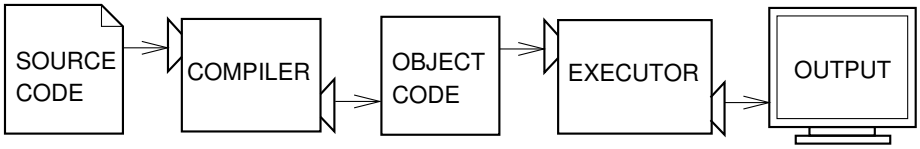
Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the

www.dbooks.org

program says. It processes the program a little at a time, alternately reading lines and performing computations.



A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



One of the nice features of OCaml is that OCaml is both a compiled and an interpreted programming language. On the one hand, OCaml can be compiled into an executable file. This makes it easier to pass along your program to non-OCaml users, which for a new, up-and-coming programming language like OCaml, is a nice feature. On the other hand, OCaml can be treated like an interpreted programming language, meaning that an OCaml script or OCaml commands can be executed by an interpreter. This is useful for finding errors in your code and testing commands and snippits of code. There are two ways to use the interpreter: **interactive mode** and **script mode**. In interactive mode, you type OCaml commands into the **toplevel** and the interpreter prints the result:

```
# 1+1;;
- : int = 2
```

The octothorp, `#`, is the **prompt** the interpreter uses to indicate that it is ready. The double semicolons indicates the end of a line. If you type `1 + 1;;`, the interpreter replies `- : int = 2`.

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, OCaml scripts have names that end with `.ml`.

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

1.2 The OCaml Toplevel

Let's take a moment to explain how to use the OCaml toplevel. The toplevel works a little differently depending on your operating system. I will cover using the toplevel in Windows (7) and Ubuntu (10.04 Karmic). [1]

One thing you will need to use are OCaml commands known as **directives**. Directives tell the interpreter or compiler to do specific tasks, like load code from a file or exit. A list of some directives can be found at `http://caml.inria.fr/pub/docs/manual-ocaml/manual023.html`.

[1] Mac users are on their own.

1.2.1 Ubuntu (10.04)

First off, you must install the ocaml package and its dependencies. Open the terminal and type:

```
sudo apt-get install ocaml
```

It will then likely ask you for your admin password. After the installation is complete, you can open the OCaml toplevel simply by typing `ocaml` at the command line.

Once in the toplevel, you can open a script by using the `#use "`*filename*`";;` directive:

```
#use "myscript.ml";;
```

Note that the toplevel's **current directory** [2] will be the current directory when you ran it within the terminal. You can change your current directory using the `#cd ``dirname''';;` directive.

1.2.2 Windows (7)

First you must install OCaml. Go to the OCaml home page (`caml.inria.fr`) and go to the Download page. Download the appropriate distribution - if you don't know which one to choose, go with the Windows binary "for the port based on the Microsoft toolchain." Run the installer and follow the directions therein.

You can now run an OCaml toplevel from the windows command line by typing `ocaml`. Alternatively, OCaml has a very pretty toplevel with some nice features which you can open by running "Objective Caml" from Windows ¿ All Programs ¿ Objective Caml, or open the same program by searching from the windows start menu.

Once in the toplevel, you can open a script by using the `#use "`*filename*`";;` directive:

```
#use "myscript.ml";;
```

Note that the toplevel's **current directory** [3] will be the current directory when you ran it within the terminal. You can change your current directory using the `#cd ``dirname''';;` directive.

1.3 What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input: Get data from the keyboard, a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

[2] The **current directory** is the directory in which ocaml will look to find files to run, outside of the standard ocaml directories.

[3] The **current directory** is the directory in which ocaml will look to find files to run, outside of the standard ocaml directories.

www.dbooks.org

conditional execution: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

That may be a little vague, but we will come back to this topic when we talk about **algorithms**.

1.4 What is debugging?

Programming is error-prone. For whimsical reasons, programming errors are called **bugs**[4] and the process of tracking them down is called **debugging**.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.4.1 Syntax errors

OCaml can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a **syntax error**.

In English readers can tolerate most syntax errors, which is why we can read the poetry of e. e. cummings without spewing error messages. OCaml is not so forgiving. If there is a single syntax error anywhere in your program, OCaml will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

1.4.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.4.3 Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

[4] It's not actually true that it's named after a moth found in a computer relay, though that does make for a rather amusing story. For more information on etymology, check out `http://en.wikipedia.org/wiki/Software_bug`.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it generally requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.4.4 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users' Guide* Beta Version 1).

Later chapters will make more suggestions about debugging and other programming practices.

1.5 Formal and natural languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

Formal languages tend to have strict rules about syntax. For example, $3+3=6$ is a syntactically correct mathematical statement, but $3+=3\$6$ is not. H_2O is a syntactically correct chemical formula, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3+=$

www.dbooks.org

3$6 is that $ is not a legal token in mathematics (at least as far as I know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement $3+ = 3\$6$ is illegal because even though $+$ and $=$ are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

Exercise 1.1 Write a well-structured English sentence with invalid tokens in it. Then write another sentence with all valid tokens but with invalid structure.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The penny dropped," you understand that "the penny" is the subject and "dropped" is the predicate. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a penny is and what it means to drop, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are some differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The penny dropped," there is probably no penny and nothing dropping[5]. Formal languages mean exactly what they say.

People who grow up speaking a natural language—everyone I know—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

[5] This idiom means that someone realized something after a period of confusion.

1.6 The first program

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words, "Hello, World!" In Ocaml, one version looks like this:

```
print_string ''Hello, World!\n'';;
```

This is an example of a **print statement**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is:

```
Hello, World!
- : unit = ()
```

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result. The "
n" is an **escape character** [6], specifically the newline character, which tells the output to move to a new line.

There's another, more general print statement called "printf," which lets you do more than $print_string$. Printf comes from it's own **module**, called Printf, meaning you call printf by typing "Printf.printf." Modules are extensions of OCaml that often contain useful functions. Printf is a standard module, meaning it comes prepackaged with most OCaml installations. You can look up printf on your own if you're interested.

Alternatively, you can import the module by using the `#use` directive:

```
# #use "printf.ml"
```

And then call printf simply as `printf`. However, importing a module generally isn't advised unless you're really going to use functions from that module a lot.

Really, we could get simply and just type "Hello, World!" and get the result "`- : string = ''Hello, World!''`."

Some people judge the quality of a programming language by the simplicity of the "Hello, World!" program. By this standard, OCaml does [7] about as well as possible.

1.7 Functional, as opposed to...?

You may have noticed that this book is entitled "Think OCaml: How to Think Like a (Functional) Computer Scientist." No, this is not saying that you'll be passable-but-not-great, at programming. Rather, it's a particular paradigm of programming. **Functional Programming** is programming where the function is a first-class citizen. Now, this is of course a silly definition, but it's the best we can really do for now.

Another popular programming paradigm is **Object-Oriented**. Languages that are object-oriented, such as C++ and Python, focus rather on having "objects" and changing them. "Object" is really kind of a blanket term for all kinds of different, often custom, ways of storing data.

[6] An escape character is a special character within a string denoted by a "whack" ("
") followed by a character or series of characters that results in a special character, such as a new line.

[7] Or can do, anyway. There are about as many different OCaml "Hello, World!" programs as there are OCaml programmers, all arguably equally valid.

www.dbooks.org

OCaml is mostly a functional programming language. However, OCaml has support for some level of object-oriented programming, unlike it's predecessor Caml. The "O" in OCaml stands for "Objective".[8]

1.8 Debugging

It is a good idea to read this book in front of a computer so you can try out the examples as you go. You can run most of the examples in interactive mode, but if you put the code into a script, it is easier to try out variations.

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the "Hello, world!" program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell $print_string$ wrong?

This kind of experiment helps you remember what you read; it also helps with debugging, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent or embarrassed.

There is evidence that people naturally respond to computers as if they were people[9]. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people.

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a debugging section, like this one, with my thoughts about debugging. I hope they help!

1.9 Glossary

problem solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like OCaml that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute; also called "machine language" or "assembly language."

portability: A property of a program that can run on more than one kind of computer.

[8] "Caml" once stood for "Categorical Abstract Machine Language" but it doesn't any more. Now it just represents an even-toed ungulate with humps on its back.

[9] See Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program written in a high-level languag into a low-level language all at once, in preparation for later execution.

source code: A program in a high-level language before being compiled.

object code: The output of the compiler after it translates the program.

executable: Another name for object code that is ready to be executed.

prompt: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

script: A program stored in a file (usually one that will be interpreted).

interactive mode: A way of using the Python interpreter by typing commands and expressions at the prompt.

script mode: A way of using the Python interpreter to read and execute statements in a script.

toplevel: The interface you interact with when working with the interpreter.

program: A set of instructions that specifies a computation.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

debugging: The process of finding and removing any of the three kinds of programming errors.

syntax: The structure of a program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

exception: An error that is detected while the program is running.

semantics: The meaning of a program.

semantic error: An error in a program that makes it do something other than what the programmer intended.

natural language: Any one of the languages that people speak that evolved naturally.

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

parse: To examine a program and analyze the syntactic structure.

print statement: An instruction that causes the OCaml interpreter to display a value on the screen.

www.dbooks.org

1.10 Exercises

Exercise 1.2 Use a web browser to go to the OCaml website `caml.inria.fr`. This page contains information about OCaml (and Caml Light) and links to OCaml-related pages, and it gives you the ability to search the Ocaml documentation.

Note that OCaml, like all good things, comes from France (hence the .fr in the url).

Exercise 1.3 Open the OCaml toplevel and create a "Hello, World!" program in interactive mode.

Now, create a script called "hello.ml" that is a "Hello, World!" program and run it.

Experiment with several different methods of writing this program. In particular, write at least one method using the Printf module. Use the OCaml documentation online to understand enough of this module to do so.

Chapter 2

Variables and Expressions

2.1 Values and types

A **value** is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are `1`, `2`, and `"Hello, World!"`.

These values belong to different **types**: `2` is an integer, and `"Hello, World!"` is a **string**, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in double quotation marks.

You've already seen `print_string`. There's also a print statement for integers:

```
# print_int 4;;
4- : unit = ()
```

If you are not sure what type a value has, the interpreter will tell you.

```
# let a = 1;;
val a : int = 1;;
# let foo = "Hello, World!"
val foo : string = "Hello, World!"
```

Not surprisingly, strings belong to the type `string` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called **floating-point**.

```
# let a = 3.2;;
val a : float = 3.2
```

There's also a separate type called "char," which is short for "character." A char is like a string, but it's only one character long (hence the name), and it's surrounded by single quotes instead of double quotes. In fact, strings are comprised of lists of chars. [1] Usually, you'll use strings, but if you really only need one character, chars use less memory.

```
# let foo = 'a';;
val foo : char = 'a'
```

[1] A note on pronunciation: most often, "char" is pronounced like in charcoal. It's less standard, but I personally prefer to pronounce it like it appears in character, since that's where it came from.

www.dbooks.org

If you attempt to make a char that's more than one character in length, you get a syntax error:

```
# let foo = 'ab';;
Error: Syntax error
```

If you enclose a number in quotes, OCaml will infer that it is a char or string, depending on the number of quotes.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in `1,000,000`. This is not a legal integer in OCaml, but it is legal:

```
>>> 1,000,000
- : int * int * int = (1, 0, 0)
```

Well, that's not what we expected at all! OCaml interprets `1,000,000` as a comma-separated sequence of integers. Nevermind the type of this object; we'll get to it later.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

There's also one more type, called **unit**, represented as (). This represents a sort of "void" or "empty" value. You'll see where this pops up, and can be useful, later.

2.2 Variables

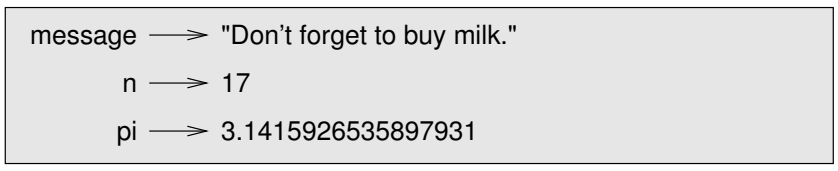
One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value. In OCaml, variables *really* refer to expressions, but don't worry about that for now.

An **assignment statement** creates new variables and gives them values:

```
# let message = "Don't forget to buy milk.";;
val message : string = "Don't forget to buy milk."
# let n = 17;;
val n : int = 17
# let pi = 3.14159265;;
val pi : float = 3.14159265
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer `17` to `n`; the third assigns the (approximate) value of π to `pi`.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the previous example:



To display the value of a variable, you can simply type it in the toplevel:

```
# n;;
- : int = 17
# pi;;
- : float = 3.14159265
```

2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`, or to denote subscripts, such as `y_0` as an initial condition. So can an apostraphe (`'`), such as `h'`.

If you give a variable an illegal name, you get a syntax error:

```
# let 17h = 17*17;;
Error: Syntax error
# let more@ = "hello";;
Error: Sytax error
# let class = 2;;
Error: Syntax error
```

At the toplevel, OCaml will be nice and even underline where the error occured for you.

`7h` is illegal because it does not begin with a letter. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of OCaml's **keywords**. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names. The full list of keywords can be found online here: `http://caml.inria.fr/pub/docs/manual-ocaml/manual044.html`.

You'll get a really interesting error if you try to use just a number as a variable:

```
# let 17 = 2;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
Exception: Match_failure ("", 8, -17).
```

That's fun. Technically, this is a semantic error causing a syntax error. This will make a little more sense later when we talk about pattern-matching.

2.4 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called **operands**.

www.dbooks.org

In OCaml, operators are **strictly typed**, meaning you can't use an integer operator for a float, or vice versa. OCaml, unlike some other programming languages, won't even infer that you meant a float when you put an int, or vice versa. This can make debugging type errors much simpler, though it seems like a pain at first.

For integers, the operators `+`, `-`, `*`, and `/` perform addition, subtraction, multiplication, and division, as in the following examples:

```
20+32   hour-1   hour*60+minute   minute/60   (5+9)*(15-7)
```

For floating-points, the operators are the same, but followed by a period, namely `+.`, `-.`, `*.`, and `/.`. There is also a floating point operator for exponentiation, namely `**`. There is no integer exponention, probably because an integer raised to a negative integer would result in a non-integer. [2]

In some other languages, `^` is used for exponentiation, but in OCaml it is used for string concatenation, meaning the two strings are added end-to-start, such as in the following example:

```
 # "Hello, "^"World!";;
- : string = "Hello, World!"
```

Don't forget to put a space somewhere so your words are separated when you concatenate.

You can change an int to a float or vice versa using the OCaml functions `float_of_int` or `int_of_float`, like so:

```
 float_of_int x;; int_of_float x;;
```

I'll leave determining which is which as an exercise to the reader.

2.5 Expressions

An **expression** is a combination of values, variables, operators and functions that returns a particular value. A value all by itself is an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17;;
x;;
x + 17;;
```

If you type an expression in interactive mode, the interpreter **evaluates** it and displays the result:

```
# 1 + 1;;
- : int = 2
```

Expressions can, and often do,[3] contain other expressions, called **subexpressions**. When evaluating the value of the expression, the OCaml interpreter will evaluate the embedded expression and substitute it in to the larger expression. In OCaml, and many other function-oriented languages, everything must be an expression. Everything up to a double semi-colon (`;;`) must be exactly one expression, with arbitrarily many sub-expressions.

Exercise 2.1 Type the following statements in the OCaml interpreter to see what they do:

[2] This is not to say that a function can't take two integers and return a float.

[3] Any expression with more than just a value or variable technically contains subexpressions.

```
5;;
let x = 5;;
x + 1;;
```

Now put the same statements into a script and run it. What is the output?

2.6 Variables as Expressions, and References

Now, this is where I admit that I lied to you a little bit.

Remember how I said that the expression "let x = 3;;" assigns the values of '3' to the variable 'x'? Well, that's not entirely true. In fact, it's not true at all.

What really happens when I say "let x = 3;;" is that the *expression* '3' is represented by 'x'. 'x' becomes kind of a shorthand way of writing '3'.

One of the side effects of this is that you can't change the value of a variable. So, really, in some sense, they shouldn't be called variables at all.

2.7 Scope

Now, I said you couldn't change the value of variables, but in some senses it looks like you can.

```
 # let x = 2;;
val x : int = 2
 # let x = 3;;
val x : int = 3;;
 # x;;
- : x = 3
```

See? The value of 'x' changed! Didn't it?

Actually, the answer is no, it didn't. What we have here is a problem of **scope**. You see, variables all act within certain bounds, or on certain levels within a program. Every time we say "let", it kind of creates a new "level" of the program, within which only certain variables can be accessed. This brings up let's counterpart, 'in'. The best way to explain 'in' is with an example.

```
 # let x = 2;;
val x : int = 2
 # let x = 3 in x+1;;
- : int = 4
 # x;;
- : int = 2
```

Now, didn't see that last bit coming, did you? When you use "let... in...;;", the value assigned by the let only exists within the expression after in. This is called a "let-in-binding".

Exercise 2.2 Play around with scope a little bit in the command line.

See if you can predict the outcome of the following:

www.dbooks.org

```
0   # let x = 2 in x+1;;
   # let x = 3 in x+1;;
   # x;;
```

```
0   # let x = 2;;
   # let y = 3 in let x = 2 in x+y;;
   # x;;
   # y;;
```

Now try them out in the interpreter and see if you were right!

2.8 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. For mathematical operators, OCaml follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1.+.1.)**(5.-.2.)` is 8.. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.
- **E**xponentiation has the next highest precedence, so `2.**1.+.1.` is 3., not 4., and `3.*1.**3.` is 3., not 27..
- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.
- Operators with the same precedence are evaluated from left to right. So in the expression `degrees /. 2. * pi`, the division happens first and the result is multiplied by `pi`. To divide by 2π, you can use parentheses or write `degrees /. 2. /. pi`.

Technically speaking, all operations in the programming language follow some sort of order of operations. However, you don't really need to know more than that parentheses override all that precedence. In fact, if you need to know more about the order of operations than the above mathematical rules, then you're not making your code clear enough and should use parentheses to make things more clear.

2.9 String operations

You cannot perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
"2"-"1"    "eggs"/"easy"    "third"*"a charm"  "you"+"me"
```

In OCaml, an operator cannot be used for two things. In some languages, for example Python, operators can be **overloaded**, meaning they do different things in different situations (i.e., '+' can be used for string concatenation). In OCaml, this is not the case.

2.10 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they start with a `(*` and end with a `*)`:

```
(* compute the percentage of the hour that has elapsed *)
percentage = (minute * 100) / 60
```

Everything between the parentheses will be ignored by the compiler. You can also put comments at the end, or even in the middle, of a line of code:

```
 # let x = 2;; (* I'm a comment! *)
val x : int = 2
 # let x (* Hi, mom! *) = 3;;
val x = 3
```

Of course, putting comments in the middle of expressions like that can be really annoying to read.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
let v = 5;;     (* assign 5 to v *)
```

This comment contains useful information that is not in the code:

```
let v = 5;;     (* velocity in meters/second. *)
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

2.11 Debugging

At this point the syntax error you are most likely to make is an illegal variable name, like `class`, which is a keyword, or `odd.job` and `hack_and_/`, which contain illegal characters.

For syntax errors, the error messages don't help much. The most common message is `SyntaxError: invalid syntax` which is not very informative.

The runtime error you are most likely to make is a "unbound value" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
 # let principal = 327.68;;
val principal : float = 327.68
 # let interest = principle *. rate;;
Error: Unbound value principle
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex`.

At this point the most likely cause of a semantic error is the order of operations. For example, to evaluate $\frac{1}{2\pi}$, you might be tempted to write

www.dbooks.org

```
 # 1. /. 2. *. pi;;
```

But the division happens first, so you would get $\pi/2$, which is not the same thing! There is no way for OCaml to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

Another semantic error you might encounter is a scope error. For example, assigning a variable for one scope and trying to use it in another:

```
 # let devil = -1;;
val a : int = -1
 # let god = 1 in god+devil;;
- : int = 0;;
 # god+devil;;
Error: unbound value god
```

The last common semantic error I'll mention now is attempting to create a variable name with a space in it, which OCaml interprets as a function definition:

```
# let bad name = 5;;
val bad : 'a -> int = <fun>
# bad 2;;
- : int = 4
# bad "Please, anything but 4!";;
- : int = 4
```

What you've done is written a function `bad` that takes one argument of any type and always returns `4`. This will make more sense when we see functions next chapter.

2.12 Glossary

value: One of the basic units of data, like a number or string, that a program manipulates.

type: A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), strings (type `str`), characters (type `char`), and the unit type (type `unit`, `()`).

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

string: A type that represents sequences of characters.

character: A type that represents a single character.

unit type: A null or empty type.

variable: A name that refers to a value. Actually, acts as shorthand for an expression.

statement: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

assignment: A statement that assigns a value to a variable.

state diagram: A graphical representation of a set of variables and the expressions they refer to.

keyword: A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `class`, and `while` as variable names.

operator: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

operand: One of the values on which an operator operates.

expression: A combination of variables, operators, and values that represents a single result value.

evaluate: To simplify an expression by performing the operations in order to yield a single value.

rules of precedence: The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

concatenate: To join two operands end-to-end.

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

2.13 Exercises

Exercise 2.3 Assume that we execute the following assignment statements:

```
let width = 17;;
let height = 12.0;;
let delimiter = '.';;
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width/2`
2. `width/.2.0`
3. `height/3`
4. `1 + 2 * 5`
5. `delimiter * 5`

Use the OCaml interpreter to check your answers.

Exercise 2.4 Practice using the OCaml interpreter as a calculator:

1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5? Hint: 392.6 is wrong!
2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

www.dbooks.org

Chapter 3

Functions

3.1 Function calls

In the context of functional programming, a **function** is a named expression that performs a computation. When you define a function, you specify the name and the expression. Later, you can "call" the function by name. We have already seen one example of a **function call**:

```
# float_of_int 32;;
- : float = 32.
```

The name of the function is `float_of_int`. The expression that follows is called the **argument** of the function. The result, for this function, is a floating-point version of the integer argument. There is, of course, the inverse function,`int_of_float`. There are other type conversion functions, too, such as `string_of_int` and `string_of_float` and their respective inverses.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

3.2 Math functions

OCaml provides most of the familiar mathematical functions, including the trig functions and their inverses, log (natural log) and base-10 log. Naturally, they all take float arguments, not integers.

```
# let ratio = signal_power /. noise_power;
# decibels = 10. *. log10 ratio;;

# radians = 0.7;;
# height = sin radians;;
```

The first example computes the base-10 logarithm of the signal-to-noise ratio.

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

www.dbooks.org

```
# let degrees = 45.;;
# let pi = 4. *. atan 1.0;;
# let radians = degrees /. 360. *. 2. *. pi;;
# sin radians;;
0.707106781187
```

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
# (sqrt 2.) /. 2.0
0.707106781187
```

3.3 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
let x = sin (degrees /. 360.0 *. 2 *. pi);;
```

And even function calls:t

```
let x = exp (log (x+.1.));;
```

Note in this example that we put parentheses around the inner function and the inner expression, to tell OCaml how to read this statement. If we just told it `exp log x+1;;` it would try to pass both `log` and `x` as two separate arguments to `exp` and then add one.

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name, or function definition, which you'll see in the next section. Any other expression on the left side is a syntax error.

```
 # let minutes = hours *. 60.0;;                (* right *)
 # let hours *. 60 = minutes;;;                 (* wrong! *)
Error: Syntax error
```

3.4 Adding new functions

So far, we have only been using the functions that come with OCaml, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called.

Here is an example:

```
let  print_lyrics () =
    print_string "I'm a lumberjack, and I'm okay. \n";
    print_string "I sleep all night and I work all day. \n";;
```

Note that we use the "let" keyord to define functions just the same as variables. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any meaningful arguments. More specifically, it takes a unit type argument, meaning when we call it we have to put empty parentheses after it. If we don't put the parentheses, then it gives us a little information about the function:

```
# print_lyrics;;
- : unit -> unit = <fun>
```

What this output means is that this is a function (`= <fun>`) which takes a unit type argument and returns a unit (`unit -> unit`). We'll see why this is useful later.

But wait! Was that a single semi-colon I saw there? What was that? I thought you used two semi-colons to end a statement? Well, sure, two semi-colons do end a statement. But if you put two semi-colons inside a function definition, that ends the entire function definiton. So if we want a function to execute one statement and then execute another statement, we terminate all the intermediate statements with single semi-colons, which is called a *sequence point*. [1] If we don't, we get an interesting syntax error:

```
# let print_lyrics () =
print_string "I'm a lumberjack and I'm okay. \n"
print_string "I work all night and I sleep all day. \n";;
Error: This function is applied to too many arguments;
maybe you forgot a `;'
```

It's trying to treat the second `print_string` as another argument in the first `print_string`! That's because we didn't give it anything to tell it that we've stopped putting in arguments to the first `print_string`. In OCaml, a new line in code is nothing more than a space. New lines and indentation just make your code easier to read and more organized. It doesn't help the OCaml compiler at all.

The syntax for calling the new function is the same as for built-in functions:

```
# print_lyrics();;
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
- : unit = ()
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
let repeat_lyrics() =
    print_lyrics();
    print_lyrics();;
```

And then call `repeat_lyrics();;`:

[1] The sequence point is actually an operator, like +. It takes a unit on the left and anything on the right and returns the right.

www.dbooks.org

```
# repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
- : unit = ()
```

But that's not really how the song goes.

3.5 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
let print_lyrics() =
    print_string "I'm a lumberjack, and I'm okay. \n";
    print_string "I sleep all night and I work all day. \n";;

let repeat_lyrics() =
    print_lyrics();
    print_lyrics();;

repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create functions. The statements inside the function do not get executed until the function is called, and the function definition generates only trivial output.

In OCaml, so you can define a function after you call it, so long as it is called and defined in the same script.

Exercise 3.1 Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see if the output changes.

Exercise 3.2 Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

3.6 Flow of execution

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, OCaml is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

3.7 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `sin` you pass a floating-point as an argument. Some functions take more than one argument.

Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

```
let print_twice bruce =
print_int bruce;
print_string " ";
print_int bruce;
print_string "\n";;
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (which must be an integer) twice, with a space in between then followed by a new line.

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`, so long as it evaluates to an integer:

```
>>> print_twice (3*4)
12 12
- : unit = ()
>>> print_twice (2+10)
12 12
- : unit = ()
```

Note that we need to tell OCaml exactly what the argument is using parentheses in this case. This is because if we omitted them it would read left to right, so the first function call would become `(print_twice 3)*4`, which doesn't make any sense, especially since `(print_twice 3)` evaluates to a unit, which you can't really multiply by four.

The argument is evaluated before the function is called, so in the examples the expressions `3*4` and `2+10` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 1
>>> print_twice michael
1 1
- : unit = ()
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

www.dbooks.org

Functions can also take other functions as arguments. Such a function is called a *Higher-Order Function*, which is sometimes abbreviated HOF.

3.8 Functions are Just Renamed Expressions

In OCaml, functions are really just expressions renamed. This means that an Ocaml function must evaluate to a value, hence why you must use sequence points within a function definition.

Using `let` within a function definition requires the in keyword:

```
let cat_twice part1 part2 =
    let cat = part1^part2^"\n" in cat^cat;;
```

This function takes two string arguments, concatenates them and adds a newline character, and then concatenates the result to itself and returns this string. Note that "in" in there: that's so that the variable definition doesn't return anything itself; it's only part of an expression which returns the result.

3.9 Scope and Functions

```
# let line1 = 'Bing tiddle ';;
# let line2 = 'tiddle bang.';;
# cat_twice line1 line2;;
Bing tiddle tiddle bang.
- : unit = ()
```

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

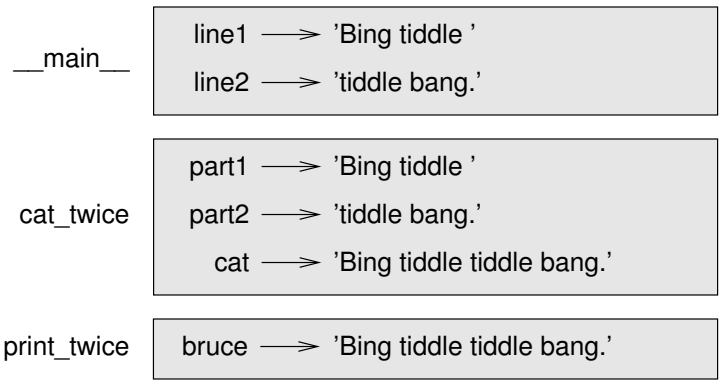
```
# print_string cat
Error: Unbound value cat
```

Parameters are also local. For example, outside `_twice`, there is no such thing as `bruce`.

3.10 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

3.11 Currying Functions

Currying functions is a very odd, but very powerful, idea in OCaml.

You've probably noticed that OCaml gives you a very interesting output after a function definition. Consider the following (somewhat trivial) function:

```
# let add a b = a+b;;
val add: int -> int -> int = <fun>
```

Effectively, what this means is that you have a function which takes two integers as input and returns an integer (in this case, the sum of the two). What's with the arrows, then? Well, this syntax was chosen for a reason.

Consider the following:

```
# add 3;;
```

What is that? If you're familiar with other programming languages, you might assume it's a syntax error: the function doesn't have enough input arguments! Well, you would be wrong. In fact, there's no error at all. That is a perfectly valid statement in OCaml:

```
# add 3;;
- : int -> int = <fun>
```

That's right, `add 3` is actually a function in it's own right. It takes an integer as an input, to which it will add three and then return the result. That's because functions of multiple input arguments in OCaml are really just chains of single-argument functions. Hence, `add a b` is really the function `add a` applied to `b`, which is really the function `add` applied to `a`, the result of which is then applied to `b`. [2] This is called '**currying**, after Haskell Curry. It is also sometimes called **partial application**.

[2] If you've never heard of currying before and that made sense to you the first time through, then you deserve a medal and you're going to love when we get to recursion.

www.dbooks.org

In fact, we can even name this function:

```
# let add3 = add 3;;
val add3 : int -> int = <fun>
# add3 2;;
- : int = 5
```

This would be effectively the same as writing:

```
# let add3 b = add 3 b;;
val add3 : int -> int = <fun>
# add3 2;;
- : int = 5
```

Except the first one is a little cleaner. In this trivial example, the difference is very slight, but when we introduce some new concepts later, you'll see why currying is so powerful.

3.12 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

3.13 Debugging

Don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same, incorrect, program over and over!

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print_int42` at the beginning of the program and run it again. If you don't see `42`, you're not running the right program!

3.14 Glossary

function: A named sequence of statements that performs some useful operation. Functions may take any parameter and return any type, including the unit type.

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

header: The first line of a function definition.

body: The sequence of statements inside a function definition.

parameter: A name used inside a function to refer to the value passed as an argument.

function call: A statement that executes a function. It consists of the function name followed by an argument list.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

local variable: A variable defined inside a function. A local variable can only be used inside its function.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

module: A file that contains a collection of related functions and other definitions.

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

flow of execution: The order in which statements are executed during a program run.

currying: Also known as partial application. The use of a function such that, instead of having multiple arguments, it is instead a chain of single-argument functions.

3.15 Exercises

Exercise 3.3 OCaml provides a standard module called `String`, which includes a useful function called `length` that returns the length of a string, so the value of

```
String.length("monje")"
```

is 5.

Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

```
# right_justify('monje')
                                                                monje
```

Exercise 3.4 Write a function that takes two strings as an argument and concatenates them with a space in between.

Now use currying with that function to define functions that append + and - to the end of the string, again with a space in between.

Finally, write a function to append a symbol x number of times.

www.dbooks.org

Exercise 3.5 This exercise[3] can be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

2. Use the previous function to draw a similar grid with four rows and four columns.

[3] Based on an exercise in Oualline, *Practical C Programming, Third Edition*, O'Reilly (1997)

Chapter 4

Program Flow

4.1 Modulus operator

The **modulus operator** works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is `mod`. The syntax is the same as for other operators:

```
# let quotient = 7 / 3;;
val quotient : int = 2
# let remainder = 7 mod 3;;
val remainder : int = 1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `x mod y` is zero, then `x` is divisible by `y`.

Also, you can extract the right-most digit or digits from a number. For example, `x mod 10` yields the right-most digit of `x` (in base 10). Similarly `x mod 100` yields the last two digits.

4.2 Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator <>, which compares two operands and produces `false` if they are equal and `true` if they are:

```
>>> 5 <> 5;;
- : bool = false
>>> 5 <> 6;;
- : bool = true
```

`True` and `False` are special values that belong to the type `bool`; they are not strings.

The <> operator is one of the **relational operators**; the others are:

www.dbooks.org

```
x <> y              # x is not equal to y
x > y               # x is greater than y
x < y               # x is less than y
x >= y              # x is greater than or equal to y
x <= y              # x is less than or equal to y
```

Although these operations are probably familiar to you, the OCaml symbols are different from the mathematical symbols.

4.3 Logical operators

There are three **logical operators**: `&&` (and), `||` (or), and `not` (exactly as it looks). The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0 && x < 10` is true only if `x` is greater than 0 *and* less than 10.

`n mod 2 = 0 || n mod 3 = 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

4.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if x > 0
then print_string "x is positive."
else print_string "x is not certain.";;
```

The boolean expression after the `if` statement is called the **condition**. If it is true, then the expression after `then` gets executed. If not, then the `else` expression is evaluated. The only caveat is that the `then` and `else` expressions must evaluate to values of the same type, both integers for example.

The else expression can be omitted, in which case it defaults to `else ()`. This can be problematic as it means the `then` clause must also evaluate to the unit type.

As with most things in OCaml, as you're probably beginning to notice, the then-expression and else-expression must evaluate to one value (i.e., be expressions). Sometimes, though, you might want to do more than one thing within those expressions. For example, you might want to display the value of something. For just that, we have `begin` and `end`.

```
if x>0
then begin
          print_int x;
          x
     end
else x+1;;
```

Note that the first line within the `begin...end` block is followed by a sequence point: anything returned by the first line will just be discarded. However, it's considered good form for it to always return the unit, and it will spew warning errors at you if it returns anything else.

4.5 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x > 3
then "x is big."
else if x = 3:
     then "x is three."
else "x is little.";;
```

There is no limit on the number of `else if` statements. However, these trees become harder to correctly interpret the more branches you put on, and the code can get very unmanagable. We'll see better ways to control program flow as we go along.

```
if choice = 'a':
then draw_a()
else if choice = 'b':
     then draw_b()
else if choice = 'c':
    then draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

Because one conditional is within the body of another conditional, it is referred to as a **nested conditional**. Nested conditionals can also occur within the `then` clauses of if statements.

```
if 0 < x
then if x < 10
print_string "x is a positive single-digit number."
```

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the above code using a single conditional:

```
if 0<x && x<10
then print_string "x is a positive single-digit number."
```

4.6 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

OCaml provides three built-in functions that get input from the keyboard, one each for getting strings, floats, and ints. They are, respectively, named `read_line`, `read_float`, and `read_int`, and all take a unit argument. When one of these functions is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes with the collected value.

```
# let input = read_line ();;
What are you waiting for?
# print_string input;;
What are you waiting for?
```

www.dbooks.org

Before getting input from the user, it is a good idea to print a prompt telling the user what to input.

```
print_string "What...is your name?\n";;
let input = read_line();;
```

You can then do something with the user's input. For example, we could have the program greet the user:

```
let greet () = print_string "What...is your name?\n";
let input = read_line() in
let greeting = "Hi "^input^"\n" in
print_string greeting;;
```

This could result in the following execution:

```
# greet ();;
What... is your name?
Monje
Hi Monje
- : unit = ()
```

where the second line is user input.

4.7 Pattern Matching

One of the coolest features of OCaml is it's ability to take some object (string, integer, anything) and match it with user-determined patterns and respond accordingly. This allows us to do all kinds of things, from making conditionals more complicated than `if... then... else` without incessant nesting, to parsing text. We'll look at complicated examples later, but for now let's look at the syntax in the abstract:

```
match object with
pattern a -> result a
| pattern b -> result b
| pattern c -> result c
...
```

The object can be absolutely anything. The pattern is something of the same type as the object, or an expression that evaluates to the same type. Note that the order is significant - OCaml will take the result of the first match, not necessarily the best one, though generally we want cases to be mutually exclusive.

The pattern needing to be the same type as the object turns out to be something of a limitation. What if we wanted to determine the sign of an integer? We could use if statements, but we really have three possibilities - negative, 0, and positive - and nesting if statements is just so ugly and hard to follow.

We might start to write a pattern like this:

```
let sign i = match i with
0 -> 0
| x -> 1 (* this will match all integers! *)
```

This example will compile, but it won't do what you want it to do. The `x` case will match all integers! We need to be able to distinguish between positive and negative integers. In order to do this, we want to **guard** these patterns. This means we use `when` to impose a condition on the pattern, like so:

```
let sign i = match i with
x when x < 0 -> -1
| 0 -> 0
| x -> 1;;
```

Note that the last line is not guarded even though we don't want all integers to return one. However, we have already eliminated all the cases we don't want to return one, so there's really no point in guarding this case. It will simply make it longer and harder to read. Additionally, if all patterns in a pattern-matching are guarded, you'll get a warning message telling you it's bad style.

You can also tell OCaml to match with anything using an _, and you can use the `as` keyword to rename a pattern. The following example is silly, but demonstrates both of these abilities:

```
let add3 i = match i with
(_ as x) -> x+3;;
```

The underscore will match anything that fits within the pattern. The `as` lets us call whatever we matched there `x` and then use it in the result side.

We'll see lots more awesome examples of pattern matching as we go along.

4.8 Glossary

modulus operator: An operator, denoted with a percent sign (`%`), that works on integers and yields the remainder when one number is divided by another.

boolean expression: An expression whose value is either `true` or `false`.

relational operator: One of the operators that compares its operands: `=`, `<>`, `>`, `<`, `>=`, and `<=`.

logical operator: One of the operators that combines boolean expressions: `and`, `or`, and `not`.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch is executed.

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

body: The sequence of statements within a compound statement.

branch: One of the alternative sequences of statements in a conditional statement.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

www.dbooks.org

4.9 Exercises

Exercise 4.1 Fermat's Last Theorem says that there are no integers a, b, and c such that

$$a^n + b^n = c^n$$

for any values of n greater than 2.

1. Write a function named `check_fermat` that takes four parameters—`a`, `b`, `c` and `n`—and that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that

   $$a^n + b^n = c^n$$

   the program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

2. Write a function that prompts the user to input values for `a`, `b`, `c` and `n`, converts them to integers, and uses `check_fermat` to check whether they violate Fermat's theorem.

Exercise 4.2 If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, it is clear that you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

> "If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can[1]."

1. Write a function named `is_triangle` that takes three integers as arguments, and that prints either "Yes" or "No," depending on whether you can or cannot form a triangle from sticks with the given lengths.

2. Write a function that prompts the user to input three stick lengths, converts them to integers, and uses `is_triangle` to check whether sticks with the given lengths can form a triangle.

[1] If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.

Chapter 5

Recursive Functions

In mathematics, a **recursive function** has the general form:

$$a_{n+1} = f(a_n, a_{n-1}, a_{n-2}...)$$

meaning that the next term in the sequence a is a function of previous terms in the sequence, generally just the last one. This process is called **recursion**.Some recursive functions will be functions of more terms than just the last term. In computer science, this idea can be used for all kinds of important calculations, and can be much simpler than non-recursive solutions - such as loops, for example.

In both mathematics and computer science, you require two things: a base case, and a recursive step. The base case is usually the simplest possible case - often $n = 0$ - and the recursive step gives the next term given the term before it. Usually in programming, we don't know the term before it when we call for it, but we can calculate it in terms of the previous case - and so on and so on recursively until the base case. The recursive step can be thought of as "wishful thinking" - if I knew the n^{th} case, then the $n+1^{th}$ case would be simple.

While usually we're looking for the n^{th} case, it's often helpful to first think of recursion in forward terms:

$$\begin{aligned}
&a_0 \\
&a_1 = f(a_0) \\
&a_2 = f(a_1) && = f(f(a_0)) \\
&a_3 = f(a_2) && = f(f(a_1))) = f(f(f(a_0)))
\end{aligned}$$

and so on and so forth.

So you can see, the third term can be evaluated directly by applying f to the second term, which is an application of f to the first term, which is itself an application of f to the base case.

If this doesn't make any sense right now, don't worry; recursion can be a hard concept to wrap your mind around at first. We'll get through a few examples and maybe it will get a little clearer.

www.dbooks.org

5.1 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. However, to define a recursive function in OCaml, you must tell the compiler you're going to do this by using `let rec` instead of just `let`. For example, look at the following function:

```
let rec countdown n =
if n <= 0
then (
print_string "Blastoff!";
print_newline())
else (
print_int n;
print_newline();
countdown (n-1); ());;
```

If `n` is 0 or negative, it outputs the word, "Blastoff!" Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument. Note the else block ends with a (), telling the compiler that it returns a unit.

What happens if we call this function like this?

```
# countdown 3;;
```

The execution of `countdown` begins with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, "Blastoff!" and then returns.
> >
> > The `countdown` that got `n=1` returns.
>
> The `countdown` that got `n=2` returns.

The `countdown` that got `n=3` returns.

And then you're back in `__main__`. So, the total output looks like this:

```
3
2
1
Blastoff!
- : unit = ()
```

As another example, we can write a function that prints a string `n` times.

```
let rec print_n n s =
    if n <= 0
    then ()
    else (
print_string s;
```

```
print_newline();
print_n (n-1) s;
());;
```

If `n <= 0` the function simply returns a unit. The flow of execution immediately returns to the caller.

The rest of the function is similar to `countdown`: if `n` is greater than 0, it displays `s` and then calls itself to display `s` $n-1$ additional times. So the number of lines of output is `1 + (n - 1)`, which adds up to `n`.

Now, this is all fun and games, but we can also use recursive functions to do real work, and return an actual value.

Let's say we wanted to add up all the integers from one to some integer n. Of course, we could use the simple analytic solution, but that would be silly.[1] A recursive solution to this problem might look like this:

```
let rec csum n =
if n = 0
then 0
else (n + csum (n-1));;
```

Exercise 5.1 OCaml wil also trace a recursive function for you (or any function, really).

Type `#trace csum` into the interpreter and call `csum 3`. See if you can interpret the result. Play around with this feature for some other functions to make sure you really understand what it's printing out.

5.2 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. The most common source of infinite recursion is forgetting to decrement when you make the recursive call:

```
let rec csum n =
if n = 0
then 0
else (n + csum n);;
```

In most programming environments, a program with infinite recursion does not really run forever. Usually, you run out of what they call "stack space", which is a function of your computer architecture.

```
# csum 10;;
Stack overflow during evaluation (looping recursion?).
```

The error can also occur if the program doesn't reach the base case for any reason. `csum`, for example, isn't particularly robust: if we give it a negative number, it explodes.

```
# csum (-2);;
Stack overflow during evaluation (looping recursion?).
```

[1] No, it wouldn't be silly. If you have a convenient analytic solution, use it, it's probably more efficient, but for the sake of pedagogy, let's go ahead and not use the analytic solution.

www.dbooks.org

`csum` will continue subtracting from -2 ad infinitum without ever reaching 0, so it will keep going until it runs out of space. This could be easily avoided by checking to make sure the argument is non-negative.

5.3 Mutually Recursive Functions

It's not really that common, but every once in a while you might want to define recursive functions that call each other. Here's a trivial example:[2] 0 is an even number, and a number is defined as even if its predecessor is odd. Naturally, we could easily just check `x mod 2` to see if x is even, but that wouldn't be pedagogically as interesting as this:

```
let rec even x =
if x = 0
then true
else odd (x-1);;
let rec odd x =
if x = 0
then false
else even (x-1);;
```

Not the prettiest solution, but it should work, right? Well, actually, no. Because of how OCaml compiles, it would need odd first to compile even, but would also need even first to compile odd. Somehow we need to tell OCaml to compile both functions at the same time. Fortunately, OCaml comes with a syntax to do this:

```
let rec even x =
if x = 0
then true
else odd (x-1)
and odd x =
if x = 0
then false
else even (x-1);;
```

5.4 Tail-end Recursion

Remember `csum`? Well, it looked like this:

```
let rec csum n =
if n=1
then 1
else n + (csum (n-1));;
```

This is a perfectly valid, and reasonably efficient solution to this problem, and the code is very easy to follow. However, there's one problem with this code: as n increases, we make proportionally more function calls to csum, each of which takes up a little bit of memory. For most situations, this is probably fine, and certainly in this case, we'd have to give it a relatively huge integer for most

[2] Thanks to Ryan Tarpine who submitted it to ocaml-tutorial.org

modern computers to run out of memory. However, for other programs, we might want to be able to write recursive functions that don't build up on that stack. But how can we do that?

The answer is called **tail-end recursion**. The general idea is to write your recursive function such that the value returned by the recursive call is what's returned by your function, i.e., there's no pending operation in the function waiting for the value returned by the recursive call. That way, the function can say, "Don't bother with me anymore, just take the answer from my recursive call as the result. You can just forget all of my state information."

Generally, in order to implement tail-end recursion we need an extra argument that accumulates the result along the way, so the inner-most recursive call has all the information from the steps before it. Going back to our summing example, you might implement it like this:

```
let rec csum_helper accum n =
if n=1
then (accum+1)
else (csum_helper (accum+n)) (n-1);;

let csum = csum_helper 0;;
```

.

Note how `csum_helper`'s return value is either the answer or simply the result of the recursive call. Hence, the computer does not need to keep this function around, it just needs to take the value from the recursive call. Also, note how I've arranged this program: because I needed an `accum` argument, but when I call the function that will always be `0`, I've seperated this into two functions, a **helper function** and a **call function**. All the work happens in the helper function, but I don't want to have to tell it that `accum` is (in the first call) just zero every time I want to call the function, so I write another function to do that for me (which I defined using currying!).

Tail-recursion is generally faster and requires less memory than standard recursion. However, it can be hard to see a tail-recursive solution the first time writing an algorithm, so it's often easier to write a standard recursion solution and then figure out a tail-recursive algorithm based on that code. Also, tail-recursive solutions are usually much less clear than standard recursion, so in cases where other people will be reading your code and you don't need to worry about the memory used, it's often better to just use standard recursion. Finally, there might not always be an obvious tail-recursive solution and it might not be the best use of your time to find an algorithm if it won't significantly improve your code's performance.

Now, you may be wondering how computers know that this function is tail-recursive. The answer is, they don't, but some compilers do. OCaml implements tail-recursion as a standard of the language, as do many other functional programming languages. However, many other programming languages, such as Python, do not implement tail-recursion. If you learn other programming languages later, make sure to find out whether or not they implement tail-recursion before using it, or else you'll gain nothing but harder-to-read code.

5.5 Debugging

The traceback OCaml displays when an error occurs contains a lot of information, but it can be overwhelming, especially when there are many frames on the stack. The most useful parts are usually:

- What kind of error it was, and

www.dbooks.org

- Where it occurred.

In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

5.6 Glossary

recursion: The process of calling the function that is currently executing.

base case: A conditional branch in a recursive function that does not make a recursive call.

infinite recursion: A recursive function that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

tail-end recursion: A recursive function whose result is either the base case or the result of a recursive call, and hence does not have an operation pending for the result of said recursive call.

5.7 Exercises

Exercise 5.2 You've probably heard of the fibonacci numbers before, but in case you haven't, they're defined by the following recursive relationship: [3]

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n+1) &= f(n)+f(n-1) \end{aligned}$$

Write a recursive function to calculate these numbers. It does not need to be tail recursive.

Exercise 5.3 The Ackermann function, $A(m,n)$, is defined[4]:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases} \tag{5.1}$$

Write a function named `ack` that evaluates Ackerman's function. Use your function to evaluate `ack(3, 4)`, which should be 125. What happens for larger values of `m` and `n`?

Exercise 5.4 A palindrome is a word that is spelled the same backward and forward, like "noon" and "redivider". Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

[3] Sometimes the specifics of the base cases looks a little different, but it all works out to be the same series.

[4] See `wikipedia.org/wiki/Sometimesthespecificsofthebasecaseslooksalittledifferent, butitallworksouttobethesameseries.Ackermann_function`.

```
let first_char word = word.[0];;

let last_char word =
let len = String.length word - 1
in word.[len];;

let middle word =
let len = String.length word - 2 in
String.sub word 1 len;;
```

Note that these function use syntax you haven't seen yet. We'll see how they work in Chapter 7.

1. Type these functions into a file named `palindrome.ml` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written `''` and contains no letters?

2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

Exercise 5.5 A number, a, is a power of b if it is divisible by b and a/b is a power of b. Write a function called `is_power` that takes parameters `a` and `b` and returns `True` if `a` is a power of `b`.

Exercise 5.6 The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder[5].

One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b, then $gcd(a,b) = gcd(b,r)$. As a base case, we can consider $gcd(a,0) = a$.

Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor. If you need help, see `wikipedia.org/wiki/Euclidean_algorithm`.

Exercise 5.7 The Hofstadter Female and Male sequences are two sequences of integers defined in terms of each other, such that:

$$F(0) = 1; M(0) = 0$$
$$F(n) = n - M(F(n-1)), n > 0$$
$$M(n) = n - F(M(n-1)), n > 0$$

The first few terms of the sequences are:

$$F: 1,1,2,2,3,3,4,5,5,...$$
$$M: 0,0,1,2,2,3,4,4,5,...$$

Write two mutually recursive functions to find the n-th term of the sequences.

[5] This exercise is based on an example from Abelson and Sussman's *Structure and Interpretation of Computer Programs*.

www.dbooks.org

Chapter 6

Algorithms

An algorithm is a general method of solving some class of problems. You use many algoirthms every day: knowing how to add two numbers, for example, requires some simple algorithm, as does any other mathematical operation. Almost every non-trivial program you will ever write will involve some sort of algorithm, so algorithm-finding is one of the fullest and most interesting branches of computer science.

6.1 Square roots

Recursion or looping is often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2}$$

For example, if a is 4 and x is 3:

```
# let a = 4.0;;
# let x = 3.0;;
# let y = (x +. a/.x) /. 2.0;;
val y : float = 2.1666666666666652
```

Which is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
# let x = y;;
# let y = (x +. a/.x) /. 2.0;;
val y : float = 2.00641025641025639
```

After a few more updates, the estimate is almost exact:

www.dbooks.org

```
# let x = y;;
# let y = (x +. a/.x) /. 2;;
val y : float = 2.00001024002621453
# let x = y;;
# let y = (x +. a/.x) /. 2;;
val y : float = 2.00000000002621459
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
# let x = 2.0;;
# let y = (x +. a/.x) /. 2;;
val y : float = 2.
# let x = y;;
# let y = (x +. a/.x) /. 2;;
val y : float = 2.
```

When `y == x`, we can stop. Here is a function that starts with an initial estimate, `x`, and improves it until it stops changing:

```
let rec sqroot a x y =
if x=y
then y
else let ynew = (x +. a/.x) /. 2.0
in sqroot a y ynew;;
```

For most values of `a` this works fine, but in general it is dangerous to test `float` equality. Floating-point values are only approximately right: most rational numbers, like $1/3$, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a `float`.

Rather than checking whether `x` and `y` are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

```
    if abs (y-x) < epsilon
```

where `epsilon` has a value like `0.0000001` that determines how close is close enough.

Exercise 6.1 Encapsulate this recursive function called `square_root` that takes `a` as a parameter, chooses a reasonable value of `epsilon`, and returns an estimate of the square root of `a`.

6.2 Algorithms

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n-1$ as the first digit and $10-n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

6.3 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more place for bugs to hide.

One way to cut your debugging time is "debugging by bisection." For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

6.4 Glossary

stuff

6.5 Exercises

Exercise 6.2 To test the square root algorithm in this chapter, you could compare it with `sqrt`. Write a function named `test_square_root` that prints a table something like this:

www.dbooks.org

```
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

The first column is a number, a; the second column is the square root of a computed with the function from Exercise 6.1; the third column is the square root computed by `sqrt`; the fourth column is the absolute value of the difference between the two estimates.

Exercise 6.3 The brilliant mathematician Srinivasa Ramanujan found an infinite series[1] that can be used to generate a numerical approximation of π:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

Write a function called `estimate_pi` that uses this formula to compute and return an estimate of π. It should use recursion to compute terms of the summation until the last term is smaller than `10.**(-15)`.

[1] See `wikipedia.org/wiki/Pi`.

Chapter 7

Strings

7.1 A string is a sequence

A string is a **sequence** of characters. You can access the characters one at a time with the bracket operator:

```
# let fruit = "banana";;
# let letter = fruit.[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`. Note that the type of `letter` is `char` not `string`.

The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
# letter
- : char = 'a'
```

For most people, the first letter of `'banana'` is `b`, not `a`. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
let letter = fruit.[0]
val letter : char = 'b'
```

So `b` is the 0th letter (“zero-eth”) of `'banana'`, `a` is the 1th letter (“one-eth”), and `n` is the 2th (“two-eth”) letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
# let letter = fruit[1.5]
Error: This expression has type float but an expression was expected of type
 int.
```

Note that `fruit.[i]` is the same as typing `String.get fruit i`.

Strings are mutable; you can change the value of characters within the string using the `set` function:

www.dbooks.org

```
# fruit.[1]<-'b';;
- : unit = ()
# fruit;;
- : string = "bbnana"
```

Note that this is equivalent to writing `String.set fruit 1 'b'`. If you try to set a character outside of a string, then you'll get an exception error.

7.2 `String.length`

Many useful string operations can be found in the `String` module. `String.length` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
# let len = String.legnth fruit;;
# let last = fruit.[length];;
Exception: Invalid_argument "index out of bounds"
```

The reason for the `Exception` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
# let last = fruit[len-1]
val last: char = 'a'
```

7.3 Substrings

One thing you'll want to be able to do with strings is select **substrings**, or sections of strings that are themselves strings. We saw an example of this before in the palindromes exercise:

```
let middle word =
let len = String.length word - 2 in
String.sub word 1 len
```

In this example, we're extracting the substring of all the characters between the first and last character.

7.4 String Traversal

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with recursion and substrings:

```
let rec traverse_print s =
print_char s.[0]; print_newline();
let len = String.length s in
if len=1
then ()
else traverse_print (String.sub s 1 (len - 1));;
```

This function traverses the string and displays each letter on a line by itself. It does this by using a recursive loop that prints the first letter than recursively calls itself on the substring that is the string without the first letter.

Exercise 7.1 Write a function that takes a string as an argument and displays the letters backward, one per line.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
let prefixes = "JKLMNOPQ";;
let suffix = "ack";;

let rec ducklings prefixes suffix =
let prefix = String.sub prefixes 0 1 in
print_string (prefix^suffix); print_newline();
let len = String.length prefixes in
if len == 1
then ()
else ducklings (String.sub prefixes 1 (len-1)) suffix;;
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
- : unit = ()
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

Exercise 7.2 Modify the program to fix this error.

7.5 Searching

What does the following code do?

www.dbooks.org

```
let rec find_helper ctr word letter =
if word.[0]=letter
then ctr
else let len = String.length word in
if len = 1
then -1
else
let ctr = ctr+1 in
let word = String.sub word 1 (len-1) in
find_helper ctr word letter;;

let find = find_helper 0;;
```

In a sense, `find` is the opposite of the `.[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`.

If the character doesn't appear in the string, the function returns `-1`.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is a called a **search**.

Exercise 7.3 Modify `find` so that it finds the last instance of the letter, rather than the first.

Exercise 7.4 Modify `find` so that it has a third parameter, the index in `word` where it should start looking.

Exercise 7.5 Write a function `count` that takes a string input and a character input and counts the number of times that character appears and returns that value.

7.6 String comparison

The relational operators work on strings. To see if two strings are equal:

```
if word = "banana":
    then print_string  "All right, bananas. \n";;
```

Other relational operations are useful for putting words in alphabetical order:

```
let is_bananas word = print_string
(match word with
"bananas" -> "Your string is bananas! \n"
| w when w<"bananas" -> word^" is less than bananas. \n"
| _ -> word^" is more than bananas. \n");;
```

OCaml does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

```
# is_bananas "pineapple";;
pineapple is more than bananas.
- : unit = ()
# is_bananas "Pineapple";;
Pineapple is less than bananas.
- : unit = ()
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

7.7 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. For debugging this kind of error, first try printing the values of the indices immediately before the line where the error appears. Now when you run the program again, you'll get more information:

7.8 Glossary

sequence: An ordered set; that is, a set of values where each value is identified by an integer index.

item: One of the values in a sequence.

index: An integer value used to select an item in a sequence, such as a character in a string.

slice: A part of a string specified by a range of indices.

empty string: A string with no characters and length 0, represented by two quotation marks.

immutable: The property of a sequence whose items cannot be assigned.

traverse: To iterate through the items in a sequence, performing a similar operation on each.

search: A pattern of traversal that stops when it finds what it is looking for.

counter: A variable used to count something, usually initialized to zero and then incremented.

method: A function that is associated with an object and called using dot notation.

invocation: A statement that calls a method.

7.9 Exercises

Exercise 7.6 Read the documentation of the String module at `http://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html`. You might want to experiment with some of them to make sure you understand how they work. `index` and `iter` are particularly useful.

Exercise 7.7 ROT13 is a weak form of encryption that involves "rotating" each letter in a word by 13 places[1]. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' shifted by 3 is 'D' and 'Z' shifted by 1 is 'A'.

Write a function called `rotate_word` that takes a string and an integer as parameters, and that returns a new string that contains the letters from the original string "rotated" by the given amount.

For example, "cheer" rotated by 7 is "jolly" and "melon" rotated by -10 is "cubed".

[1] See `wikipedia.org/wiki/ROT13`.

www.dbooks.org

You might want to use the built-in `String.iter` function, and you'll probably find at least a couple of the functions at `http://gallium.inria.fr/~remy/poly/ocaml/htmlman/libref/Char.html` useful.

Potentially offensive jokes on the Internet are sometimes encoded in ROT13. If you are rather bored by not easily offended, find and decode some of them.

Chapter 8

Lists

8.1 A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`) and separate them by semi-colons:

```
[10; 20; 30; 40]
['crunchy frog'; 'ram bladder'; 'lark vomit']
```

The first example is a list of four integers, and has type `int list`. The second is a list of three strings, and has type `string list`. The elements of a list must be the same type. Note that these elements could be more lists:

```
[[10; 20]l[30;40]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`. The empty list is, unsurprisingly, polymorphic and so has type `'a list`.

As you might expect, you can assign list values to variables:

```
# let cheeses = ["Cheddar"; "Edam"; "Gouda"; "Tomme de Savoie"];;
# let numbers = [17; 123];;
# let empty = [];;
```

Another way to construct a list is using the **cons** operator, `::`, which appends an element to the beginning of a list. The following are all equivalent:

```
['a';'b';'c']
'a'::['b';'c']
'a'::'b'::['c']
'a'::'b'::'c'::[]
```

However, the following are *not* equivalent to the above:

www.dbooks.org

```
['a';'b']::['c']
'a'::'b'::'c'
['a';'b';'c']::[]
```

Test them in the interpreter and see what these do.

It's very common in functional programming to refer to the first element of a list as the **head** and the rest of the elements as the **tail**. This concept is so common, in fact, that ocaml has a built-in function to extract them:

```
# List.hd cheeses;;
- : string = "Cheddar"
#List.tl cheeses;;
- : string list = ["Edam";"Gouda";"Tomme de Savoie"]
```

We can access any element of the list by using `List.nth <list> <index>`. Remember that the indices start at 0:

```
# List.nth cheeses 0
- : string = "Cheddar"
```

Unlike strings, lists aren't mutable.

8.2 List operations

The @ operator concatenates lists:

```
# let a = [1; 2; 3];;
# let b = [4; 5; 6];;
# let c = a@b;;
val c : int list = [1; 2; 3; 4; 5; 6]
```

If you want to make a list of lists flat (i.e., turn it into a list of just elements, you can use List.flatten <list>:

```
# let c = [[1; 2; 3]; [3; 4; 5]];;
# List.flatten c;;
- : int list = [1; 2; 3; 4; 5; 6]
```

You can also reverse a list:

```
# let forward = [1; 2; 3; 4; 5];;
# let backward = List.rev forward;;
val backward : int list = [5; 4; 3; 2; 1];
```

8.3 List iteration, mapping and folding

If you want to apply a function to every element in a list, you can use List.iter function list. For example, let's say I wanted to add print every element of a list:

```
# let f elem =
    print_int elem; print_newline()
in
  List.iter f forward;;
1
2
3
4
5
- : unit = ()
```

However, the function given to List.iter must return a unit type. If we want to do something more productive - for example, adding a constant to a list, we need to use List.map:

```
# let add3 e = e+3 in List.map add3 forward;;
- : int list = [4; 5; 6; 7; 8]
```

Now, what if we wanted to sum together the elements of a list? We could write a recursive function to add together the elements, or would could **fold** the list:

```
# List.fold_left (+) 0 forward;;
- : int = 15
```

Note that this is fold_left, which operates from left to right. There's also fold_right, which operates the opposite direction. Naturally, for addition, there's no difference when you're adding, but in some cases the result will be different.

We haven't seen the syntax for the first argument before - putting an operator between parentheses turns it into a regular function.

The second argument is the initial value - if we had used an empty list, we would have returned this value. If we instead used 2, we would have gotten the result of 17.

The last argument is the list, of course. We can encapsulate this as a function:

```
let summer = List.fold_left (+) 0;;
```

Let's break down what's going down here a little bit more. Essentially what we're doing is inserting a plus sign between each element of the list and evaluating the expression. Of course, the process is actually recursive: it adds the first element to the result of the folding the rest of the list, so on and so forth until it reaches the last element, which it adds to the initial condition.

Exercise 8.1 Now use `fold_left` to write a function that finds the product of all the elements of a list. Consider carefully - what's your initial condition here? How do you want to handle the empty list?

www.dbooks.org

8.4 Traversing a list

The most common way to traverse the elements of a list is, in case you couldn't guess by now, is with recursion. This is where list-builder notation and the idea of the head and the tail becomes useful.

```
let rec printer lst = match lst with
[] -> print_string "All done! \n"
| hd::tl -> print_string hd;
print_newline();
printer tl;;

let cheeses = ["Cheddar"; "Edam"; "Gouda"; "Tomme de Savoie"]

# printer cheeses;;
Cheddar
Edam
Gouda
Tomme de Savoie
All done!
- : unit = ()
```

Though recursive list traversal is often very useful, if you are thinking about doing it, you should first stop and think about whether or not you really want List.iter, List.map, or list folding.

8.5 List sorting

Exercise 8.2 Write a function that takes a list of integers as an argument and returns a list of the same elements in increasing numeric order.

Note: if you look at the OCaml List module, you'll find several useful functions that will make this trivial. In general, it's always a good idea to see if somebody has already coded solutions relating to your problem, but for pedagogy's sake, don't do that right now. We'll talk about some of those built-ins later.

OCaml provides a few useful built-in functions to help with list sorting. The most useful of these is, unsurprisingly, List.sort. This is an example of a higher-order function: it takes two arguments, a function to use to determine the sort order, and the list itself. [1]

The sorting function must take two arguments of the same type, whatever the type of the list to be sorted is (or polymorphic), and it must return a positive integer if the first element comes after the second element, zero if they are equal, and a negative integer if the first element comes first.

Let's look at an example of sorting a list of integers in increasing order:

[1] Experienced programmers will be interested to know that this built-in function uses a merge sort and is not tail-recursive.

```
let sort_help e1 e2 = if e1>e2
then 1
else if e1=e2
then 0
else -1;;

let sorter lst = List.sort sort_help lst;;

let l1 = [4; 6; 8; 3; 5; 1; 2];;

# sorter l1;;
- : int list =[1; 2; 3; 4; 5; 6; 8]
```

You may have noticed, if you loaded the code above, that sort_help and sorter are both polymorphic. This makes sense, since > is polymorphic. In fact, the sort we just wrote works on any basic data type:

```
let cheeses = ["Gouda"; "Cheddar"; "Tomme de Savoie"; "Edam"];;

# sorter cheeses;;
- : string list = ["Cheddar"; "Edam"; "Gouda"; "Tomme de Savoie"]
```

Our sort function sorted the list into increasing alphabetical order! However, as you might expect from before, lower case doesn't work quite right:

```
let cheeses = ["Gouda"; "Cheddar"; "Tomme de Savoie"; "edam"];;

#sorter cheeses;
- : string list = ["Cheddar"; "Gouda"; "Tomee de Savoie"; "edam"]
```

As a computer, that makes perfect sense, but as a human, it's not quite what we'd usually expect. If you want to sort a list alphabetically, you'll have to do a little more work. Often it's easiest just to make everything lower case.

OCaml also has a built-in function compare that works on most data types for sorting.

This is where **anonymous functions** can be useful. An anonymous function is just like any other function, but we don't assign it a name. However, wherever we would put a function call, we can put an anonymous function to achieve the same effect. For example, if we want to square a number, we could write a function square that would do that for us, but if we only need to do it once, we can just write an anonymous function:

```
# (fun x -> x*x) 3;;
- : int = 9
```

It may not be immediately apparent why this is useful, but you'll see. In the meantime, note the syntax - fun lets the compiler know that this is a function. Then, the left-hand side of the arrow operator (of pattern-matching fame) is the arguments taken by the function, and the

www.dbooks.org

right-hand side is what the function returns. We can use this to define named functions, by the way:

```
# let square = fun x -> x*x;;
val square : int -> int = <fun>
```

This is completely equivalent to our previous function-defining syntax. It's not normally used, but is actually kind of a good way of thinking about what's going on when you define a function.

We can use anonymous functions to within defining sorter to make our code a little more concise:

```
let sorter lst = List.sort (fun e1 e2 -> e1-e2) lst;;
```

8.6 Lists and Recursion

Recursion and lists go together like bagels and cream cheese, but less fattening.

One thing we might want to do is use a loop to build up a list. Let's say you wanted to write a function range that takes two integer arguments and returns a list of all integers in between the two arguments, inclusive. You might write a function that looks a bit like this:

```
let rec range_helper m n accum =
if m=n
then n::accum
else (range_helper m (n-1) (n::accum));;

let range m n =
if m>n
then range_helper n m []
else range_helper m n [];;
```

We can also use recursion over the elements of a list to return a result. For instance, we could add up all the elements of a list:[2]

```
let rec sum_help lst accum =
match lst with
[] -> accum
| h::t -> sum_help t (h+accum);;

let summer lst =
sum_help lst 0;;
```

Both of these patterns are very important in functional programming in OCaml.

[2] You could, of course, also use list folding to do this.

8.7 Debugging

8.8 Glossary

list: An ordered sequence of values.

element: One of the values in a list (or other sequence), also called items.

nested list: A list that is an element of another list.

list traversal: The sequential accessing of each element in a list.

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

map: A processing pattern that traverses a sequence and performs an operation on each element.

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

8.9 Exercises

Exercise 8.3 Write a function that takes a list of numbers and returns the cumulative sum; that is, a new list where the *i*th element is the sum of the first $i+1$ elements from the original list. For example, the cumulative sum of `[1, 2, 3]` is `[1, 3, 6]`.

Exercise 8.4 Write a function that removes the nth element of a list.

Exercise 8.5 Write a function that removes the first element of a list that matches a provided argument. Modify it to remove all matching elements of the list.

Exercise 8.6 Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the relational operators <, >, etc.

For example, `is_sorted [1; 2; 2]` should return `True` and `is_sorted ['b'; 'a']` should return `False`.

Exercise 8.7 Write a function to turn a list of characters into a string. Write another function to do the reverse task.

Exercise 8.8 Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams.

Exercise 8.9 The (so-called) Birthday Paradox:

1. Write a function called `has_duplicates` that takes a list and returns `True` if there is any element that appears more than once. It should not modify the original list.
2. If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the `Random` module.

www.dbooks.org

You can read about this problem at wikipedia.org/wiki/Birthday_paradox.

Exercise 8.10 Write a function called remove_duplicates that takes a list and returns a new list with only the unique elements from the original. Hint: they don't have to be in the same order.

Exercise 8.11 To check whether a word is in a word list, you could use the List.exists function, but it would be slow because it searches through the words in order.

Because the words are in alphabetical order, we can speed things up with a bisection search (also known as binary search), which is similar to what you do when you look a word up in the dictionary. You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, then you search the first half of the list the same way. Otherwise you search the second half.

Either way, you cut the remaining search space in half. If the word list has 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

Write a function called bisect that takes a sorted list and a target value and returns true if it is in the list and false if it's not. Bonus points if the funciton is polymorphic.

In trying to do this, you may find that it's exceptionally difficult with a list. We'll see better data structures for doing this in a moment.

Exercise 8.12 Two words are a "reverse pair" if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list.

Exercise 8.13 Two words "interlock" if taking alternating letters from each forms a new word[3]. For example, "shoe" and "cold" interlock to form "schooled."

1. Write a program that finds all pairs of words that interlock. Hint: don't enumerate all pairs!
2. Can you find any words that are three-way interlocked; that is, every third letter forms a word, starting from the first, second or third?

[3]This exercise is inspired by an example at puzzlers.org.

Chapter 9

Case Study: Regular Expressions

Regular expressions exist in a lot of programming languages, and OCaml is no exception. **Regular expressions**, often abbreviated as **regex** or **regexp**, are a powerful way of processing text, used for anything from find and replace to punctuation stripping to removing duplicate words.

9.1 File I/O

First, we're going to want a way of reading and writing files for this chapter. Basic file input/output (I/O, sometimes just IO) is pretty simple in OCaml. Let's say you have a plain-text file named test.txt in your current directory, which reads, ``Hi!
n This is a test file!'' and you want to read it into OCaml. To do so, you need to open what's called a **channel**. Channels come in two varieties, input, or read, and output, or write. If we want to read test.txt, we want an input channel:

```
let chan_in = open_in "test.txt";;
val chan_in = in_channel <abstr>
```

We can now read in the text of that file. There are two primary read functions, one to read a character at a time, and another to read an entire line at a time. There

```
# input_char chan_in;;
- : char = 'H'
# input_char chan_in;;
- : char = 'i'
# input_char chan_in;;
- : char = '!'
# input_char chan_in;;
- : char = '\n'
# input_line chan_in;;
- : string = "This is a test file!"
```

www.dbooks.org

```
# input_char chan_in;;
Exception: End_of_file.
```

As useful as it is to read a file a character or a line at a time, sometimes we might want to move to a specific position within a file to be able to read it there. In particular, we might want to move back to the beginning of a file. pos_in returns the current position within a file, with the start of the file being position zero, of course. seek_in goes to a given location.

```
# pos_in chan_in;;
- : int = 25;;
# seek_in chan_in 0;;
- : unit = ()
# pos_in chan_in;;
- : int = 0
# input_line chan_in;;
- : string = "Hi!"
```

We can determine the length of the channel using in_channel_length:

```
in_channel_length chan_in;;
- : int = 25
```

We may also want to write to a file. For that, we want to create an output file:

```
# let chan_out = open_out "test2.txt";;
val out_chan : out_channel <abstr>
```

If the file you specify does not exist, it is created, and if it does, it is truncated.[1]

We can write to the channel using output_char and output_string:

```
# output_char chan_out 'a';;
- : unit = ()
# output_string chan_out "bcde";;
- : unit = ()
```

We can move position in an output channel, just like in input channels:

```
# pos_out chan_out;;
- : int = 5
# seek_out chan_out;;
- : unit = ()
# output_string "fg";;
- : unit = ()
```

As you may have already noticed, nothing has actually happened to the file yet. Nothing will, until you close the file:

[1] There are ways to open files in other modes, like append.
For more information, look up open_out_gen in the Pervasives module in the OCaml documentation.

```
# close_out chan_out;;
- : unit = ()
```

You can now view the file. Note that it reads ``abcfg'': when we went back in the file and wrote more, it overwrote, it didn't insert. You can also close input files with close_in.

There's also a built-in output channel: the screen. It is named stdout, and you can write to it just like any other output channel.

```
output_string stdout "Hello, world!\n"
Hello, world!
- : unit = ()
```

9.2 The Str module

The Str module is a very common module in OCaml coding, and it allows for a lot of functionality that experienced programmers would expect to deal with strings.

However, Str is not a standard module, in the way that List and String are. In order to use it, we must first load it:

```
# #load "str.cma";;
```

We can now use functions and types defined within the module just as we would normally expect. Note that this is different than if we had called #use. We still call it's functions by calling Str.function. We can now #use it if we so desire.

9.3 Regular Expressions

A regular expression is, at its simplest, a sequence of symbols with particular meanings that are used to *match* certain strings or substrings. This will make much more sense with examples. The point however, is that we can match certain strings within a document, which we can use to find information, or check that input is of a certain form, et cetera.

Perhaps the simplest regular expression is just a string. The regular expression ``a'' will match any string or substring of exactly the form ``a''. This might be useful if we were looking for all the uses of the word ``a'' within a document, though we must be careful because it will also match any time the letter ``a'' appears *within* a word. It also won't generally match ``A,'' which we probably do want to match. This is, perhaps, not a compelling example; it would not be particularly difficult to find all instances of the word ``a'' without using a regular expression, especially considering all the built-in search functionality of most any text editor these days.

www.dbooks.org

Fortunately, regular expressions allow us to do much cooler things. For example, we can detect the ends of words using the special ``\'' character. If we want to find words that end in ``ly'', for example, which might make a simple method of finding most, but not all, adverbs, we can use the regular expression ``ly\.''

Lots of useful functionality in regular expressions comes from some of its operators. Most of the notable regular expression operators are postfix operators, meaning they operate on the character before them. For example, the ``*'' token matches the preceding expression any number of times, including zero. For example, the regular expression ``pbb*t'' would match any of ``pbt,'' ``pbbt,'' or ``pbbbbbbt.'' But that's just rude. You can also group expressions using ``
(...
)'', and then a postfix operator after the grouping will be applied to the entire grouping, i.e., ``
(aa
)*'' will match any sequence with an even number of a's.

There are also character classes, which match a single character out of a set of characters, denoted by square brackets. If we want to match a single digit of a number, we can use the regular expression ``[0123456789],'' or ``[0-9]'' for short. You can also negate a character set by making the first character ``^'', which will make the character class match any single character *not* listed. ``[^0-9]'', for example, will match any character that is not a numeral.

Some systems of regular expressions get more complicated than this,[2] but this pretty much wraps it up for regular expressions in OCaml. The syntax of regular expressions in OCaml is summed up in Table 9.3. They are also summarized in the ocaml Str module documentation.

Token	Description
\b	Matches a word boundary.
.	Matches any character except newline
\^	Matches the beginning of a line or string
$	Matches the end of a line or string.
*	Postfix Operator. Repeats the preceding character, zero, one, or any number of times.
+	Postfix Operator. Repeats the preceding character one or more times.
?	Postfix Operator. Repeats the preceding character zero or one times.
|	Infix operator. Matches either the expression before the bar or the expression after the bar,
[...]	Defines a character class. It matches exactly one character, of any within the class. You can
(...)	Groups a subexpression.
\1	Matches the text matched by the first subexpression. (\2 matches the second, and so on up to \
\\	Matches a special character, i.e., one of \$\^.*+?[].

Note that when entering any of the characters with a ``
'' in an OCaml string, you will have to double the backslash so that it doesn't try to make it into an escape sequence.

[2] Technically speaking, anything that is still technically regular can be built out of these tokens, but some systems of "regular" expressions include some useful functionality that is not, strictly speaking regular.

Exercise 9.1 Write a regular expression to match doubled words, ex., "the the". Spend some time to think through what should and shouldn't get matched, considering cases of capitalization or different punctuation.

9.4 Compiling Regular Expressions

In order to use regular expressions in OCaml, you must first compile them. There are two functions to compile regular expressions, regexp and regexp_case_fold. The difference is only that the latter ignores the difference between capital letters and lower case letters. Otherwise, the look exactly the same, so I'll just use regexp for now:

```
# let r = Str.regexp "a+";;
val r : Str.regexp = <abstr>
```

There's also quote and regexp_string, both of which take a string and return a regular expression that matches exactly that string, the difference being that quote returns it as a string and regexp_string returns it as a compiled regular expression:

```
# let q = Str.quote "a+";;
val q : string = "a\\+"
# let r2 = Str.regexp_string "a+";;
val r2 : Str.regexp = <abstr>
```

Note that this will match ``a+'', not ``aaa''. There's also a regexp_string_case_fold, that is regexp_string but case-insensitive.

9.5 Using Regular Expressions: Search

Now that we have our compiled regular expression objects, we can use them to do work for us.

First, we can check whether or not a substring matches a regular expression:

```
# let s = "aaaa";;
# Str.string_match r s 0;;
- : bool = true
```

Str.string_match is a function which takes a regular expression, a string, and an int, and determines whether or not a substring starting at the position specified[3] matches the regular expression. Note that this doesn't test the entire string, only searches for the (longest) substring starting at that position:

[3] Strings are indexed from 0!

www.dbooks.org

```
# let s2 = "baaa";;
# let s3 = "aaab";;
# Str.string_match r s2 0;;
- : bool = false
# Str.string_match r s2 1;;
- : bool = true
# Str.string_match r s3 0;;
- : bool = true
```

We can determine the string matched by string_match by using matched_string:

```
# Str.string_match r s3 0;;
# Str.matched_string s3;;
- : string = "aaa"
```

matched_string takes one argument: the same string passed to string_match,[4] and returns the substring which matches the regular expression. Note that shorter substrings ``aa'' and ``a'' would also match the regular expression, but it returns ``aaa''.

There are lots of other useful string-searching functions available; look them up in the OCaml documentation online.

Exercise 9.2

9.6 Using Regular Expressions: Replacement

You can also use regular expressions to aid in string replacement.

global_replace is a function that takes three arguments, the first being a regular expression. Then, wherever the regular expression matches in the last argument, it replaces it with the template defined by the second argument. That template can be, at its most basic, a string. First, let's say you wanted to replace all the instances of the letter ``a'' with the letter ``o'':

```
# let r = Str.regexp "a";;
# let s = "Ya dawg";;
# let s2 = Str.global_replace r "o" s;;
s2 = "Yo dowg";;
```

but that's not right either. We could replace only the first letter using replace_first, which would give us ``Yo dawg'', which is clearly the correct answer.

We can do even more with global_replace. The replacement template can contain
1,
2, et cetera to match the corresponding group within the part of the string

[4] Doesn't mattter why, does it?

matched by the regular expression. Let's use the regular expression you wrote that matched doubled words in exercise 9.1 to remove the doubles. Chances are you wrote something that had some group that matched the text of the first word. If not, do it again.

```
# let s = "Yo yo dawg dawg. That's that and and all.";;
# let s2 = Str.global_replace r_doubles "\\1" s;;
```

Take a look at what your regular expression caught, or maybe didn't catch. Did you get rid of ``That's that''? Depending on the situation, you may or may not want to catch that. Write a version of your regular expression for both situations (you're more likely to not want to catch it, so maybe think a little harder about that one).

Once again, there are lots of other useful functions that you should take a look at in the documentation.

9.7 Using Regular Expressions: Splitting

One of the more common tasks within computer science is splitting up a string into a list of smaller strings (usually words) that are easier to process.

As a first pass, let's first break up a string into words just using a space as a delimeter:

```
# let s = "this is a list of words";;
# let r = Str.regexp " ";;
# let l = Str.split r s;;
val l : string list = ["this"; "is"; "a"; "word"; "list"]
```

The function split takes a regular expression and a string, and returns a the string broken up into a list using strings matched the regular expression as delimiters.

As always, the documentation has some more useful functions for you.

Exercise 9.3 As an exercise, write a better function to split strings into lists of words. Be sure to consider carefully how to handle things like punctuation, double spaces, tabs, and new lines.

9.8 Debugging

9.9 Glossary

www.dbooks.org

Chapter 10

Putting the O in OCaml, Part 1: Imperative programming

Thus far, everything we've learned has been (relatively) completely functional. However, OCaml is a practical language, and uses object-oriented functionality when it's, well, practical to do so. There's a lot to object-oriented programming, but for now let's just start with the basics.

The first thing we're going to do to make OCaml objective is learn how to make and manipulate dynamic variables. Technically speaking, this process is imperative programming. However, object-oriented and imperative programming are closely tied - object-oriented might be thought of as built upon imperative programming - and for the sake of this book, I'll use the two almost interchangeably.

10.1 References

In function-oriented languages like OCaml, you aren't really supposed to change the value of variables. You only use them as short-hand for expressions. If you really want to, though, you can create a **reference**, which functions a lot more like an object-oriented variable. This is what it looks like:

```
 # let my_ref = ref 100;;
val my_ref : int ref = {contents = 100}
```

The output looks a little weird, but that's because, in OCaml, references take a backseat to expressions.

Now, I said you could change the value of references, so let's see that.

```
 # my_ref := 42;;
- : unit = ()
```

Note that the operator for changing the value of a reference is `':='` instead of just `'='`. Also, note that assignment doesn't return a value, but rather

www.dbooks.org

returns a unit. In order to read the value of a reference, use the '!' operator. [1]

```
 # !my_ref;;
- : int = 42
```

This is all fine and well. But references aren't really any more useful than variables unless you have a more meaningful way to modify them. The most accessible method of modifying references is using loops.

10.2 Looping

In functional programming, we don't really like loops. We'd much rather use recursion, and you can usually use recursion instead. However, looping can occasionally be very useful, even in otherwise mostly-functional programs.

There are two primary forms of loops: the for loop and the while loop. Technically speaking, a for loop is just a particular case of a very useful while loop, but it's also a little easier to use and a little harder to mess up, so we'll start with that.

10.2.1 For loop

Let's look at the problem of adding itegers from 1 to n again. We might go about this problem by iterating through all the integers from 1 to n and adding them up as we go along. Well, this is exactly what a for loop is used for.

```
let n = 10;;
let addup = ref 0;;
for i = 1 to n do
     addup := !addup + i
done;;
print_int !addup;;
```

Note that I used a reference so that I could change the value of addup every time, and the use of the ! to ``get'' the value of addup in order to add to it. Also note that, in OCaml, whatever occurs between the ``do'' and the ``done'' must evaluate to a unit. That means for loops cannot return a value, which makes them not particularly useful. Fortunately, though, the reference assignment operator (:=) does not return a value.

This is a very object-oriented approach to solving this problem. You shouldn't like it. [2]

[1] In many other programming languages, '!' is used to mean "not." In OCaml, it's better to think of it as "get".

[2] Well, okay. Object-oriented solutions are very good for some problems. In this particular (trivial) case, I can't really say whether functional or imperative solutions are better (we'll see the function-oriented solution next chapter). There are definitely situations where one is easier, or faster, or more robust then the other.

10.2.2 While loop

While loops are a slightly more general form of for loops, that look at a condition and go while (hence the name) that condition is still true. Let's look at the above problem using a while loop:

```
let n = 10;;
let addup = ref 0;;
let i = ref 1;;
while !i <= n do
     addup := !addup + !i;
     i := !i + 1
done;;
print_int !addup;;
```

You can probably see the similarity between the two different loops.

One thing to be careful of in while loops is that the condition ceases to be true at some point. If the condition of a while loop is always true, then you will never stop looping (within the technical constraints of your machine). For example, if I hadn't made i a reference variable:

```
let n = 10;;
let addup = ref 0;;
let i = 1;;
while i <= n do
     addup := !addup + i;
     i = i + 1
done;;
print_int !addup;;
```

then the condition is always true. This is because, in the condition of the while loop, i is always equal to 1. The statement i = i+1 is actually an equality test which will always return false. This program will return a compile warning about how expressions within a while loop returning should return the unit type, and then continue to loop forever. This is known as an **infinite loop**, and they should generally be avoided.

This is all I'm going to cover for now. There's a lot more to object-oriented programming than this, of course, but we'll need a couple more things before we can really cover those topics.

10.3 Examples

www.dbooks.org

Chapter 11

Arrays

OCaml arrays are quite a bit like lists, but with some very significant differences, that make them suited to different tasks. The primary difference is that while lists are immutable, arrays are not. Arrays are also much better for selecting a random element, rather than just the first. As you might expect, most array functions are in the Array module. In general terms, arrays are a much more object-oriented data structure.

11.1 Making Arrays

In order to create an array, you can write it out as a list with vertical bars inside the square brackets, or use the make or init functions:

```
# let arr = [|6; 7; 3; 1; 2|];;
val arr : int array = [|6; 7; 3; 1; 2|]
# let arr = Array.init 5 (fun i -> i*i);;
val arr : int array = [|0; 1; 4; 9; 16|]
# let arr = Array.make 5 1;;
val arr : int array = [|1; 1; 1; 1; 1|]
```

As you can probably see from the examples, each way has different uses. If you want to specify every element, you'll want to write it out, though this is generally not advantageous in most applications of arrays. If there's a pattern to your elements, you want to init the array, which takes two arguments, the length of the array and a function that takes as an argument the index of the element (starting at 0!) and returning the element (not necessarily an integer!) that goes in that spot. If you want an array with all the same element, you'll want to make the array - the first argument is the length of the array, and the second argument is the element to fill the array. This gets more interesting when you fill it with a variable, especially a reference.

```
# let x = ref 1;;
# let xarr = Array.make 4 x;;
- : int ref array  =
```

www.dbooks.org

```
[|{contents = 1}; {contents = 1}; {contents = 1}; {contents = 1}|]
# x := 2;;
- : unit = ()
# xarr;;
- : int ref array  =
[|{contents = 2}; {contents = 2}; {contents = 2}; {contents = 2}|]
```

This is quite interesting. The array doesn't contain four ones, it contains four pointers to the object x. This can be useful, but it can also be dangerous. It is an instance of a phenomenon called **aliasing**, which I will discuss more later, but the jist is that we have many ways to refer to the same object, and we must be careful about changing that object from one way and affecting another unexpectedly.

You can also convert arrays to lists and vice versa using the Array.to_list and Array.of_list functions.

11.2 Array Operations

You can call a specific element of an array using get or an equivalent array syntax:

```
# arr;;
- : int array = [|0; 1; 4; 9; 16|]
# Array.get arr 0;;
- : int = 0
# arr.(2);;
- : int = 4
```

Again, indices start at 0.

Arrays are mutable. To change the value of an array element, you can use the set function or the assignment operator:

```
# arr;;
- : int array = [|0; 1; 4; 9; 16|]
# Array.set arr 3 5;;
- : unit = ()
# arr;;
- : int array = [|0; 1; 4; 5; 16|]
# arr.(3) <- 7
- : unit = ()
# arr;;
- : int array = [|0; 1; 4; 7; 16|]
```

Even though arrays are mutable, you can't change the length of arrays directly. There are handy ways to append arrays or take subarrays as new arrays, though.

```
# let arr1 = [|0; 1; 2|];;
# let arr2 = [|3; 4|];;
# let subarr1 = Array.sub arr1 0 2;;
```

```
val subarr1 : int array = [|0; 1|]
# let apparr12 = Array.append arr1 arr2;;
val apparr12 = int array [|0; 1; 2; 3; 4|]
# let conarr12 = Array.concat [arr1,arr2];;
val conarr12 = int array [|0; 1; 2; 3; 4|]
```

sub takes an array and two integers and returns a subarray of that array starting at the first integer argument, with the length specified in the second argument. append appends two arrays end-to-start. concat does the same thing as append for a list of arrays.

11.3 Array Iteration, Mapping, and Folding

Much like in lists, the Array module defines functions iter, map, fold_left and fold_right. They work in exactly the same way:

```
let arr = [| 1; 3; 6; 2; 5|];;

print_string "Iter results:"; print_newline();
Array.iter (fun el -> print_int el; print_newline()) arr;;

print_string "Map results:"; print_newline();
Array.map (fun el -> 2*el) arr;;

print_string "Fold results:"; print_newline();
Array.fold_left (fun el1 el2 -> el1+el2) 0 arr;;

Iter results:
1
3
6
2
5
- : unit = ()
Map results:
- : int array = [|2; 6; 12; 4; 10|]
Fold results:
- : int = 17
```

11.4 Array Sorting

You can sort arrays much like you can sort lists, using sort. However, it might not work as you expect:

```
# let arr = [| 1; 3; 6; 2; 5|];;
# let sortarr = Array.sort compare arr;;
val sortarr : unit = ()
```

Well, that's not right. We've sorted it into oblivion! But, wait, what's arr again?

www.dbooks.org

```
# arr;;
- : int array = [|1; 2; 3; 5; 6|]
```

let max arr = Array.sort compare arr; let n = Array.length arr - 1 in arr.(n);; arr is sorted! This is the first example we've seen of **side effects**, which are very common in object-oriented programming, but do not exist in functional programming. Array.sort doesn't return a sorted array. In fact, it doesn't really return anything - though that's not always the case in a function with side effects. The important part is that the function causes something ***else*** to happen. In this case, it sorts the array.

Sometimes side effects aren't the main point. For example, let's say we wanted the largest element of an array. We might define a function to do it like this:

```
let max arr =
Array.sort compare arr;
let n = Array.length arr - 1 in
arr.(n);;
```

This function would have the (presumably unintended) consequence of changing the array arr (specifically, it would sort it).

11.5 Array Traversal

Seeing as arrays are somewhat object-oriented data types, it makes sense to talk about traversing them in object-oriented ways. In general, you can, and should, use the iter and map functions. But this is a chance to show that looping can be useful.

Let's rewrite our max function in a way that's more object-oriented - using a while loop:

```
let max arr =
let curmax = ref arr.(0) and i = ref 1 in
while !i < Array.length arr do
if (arr.(!i) > (!curmax)) then curmax := arr.(!i);
i := !i + 1
done;
!curmax;;
```

This function traverses the loop, looking at each element, processing it, then moves on to the next element. Note that this isn't as pretty as it could be - there are lots of exclamation points and the like. This is because OCaml is, first and foremost, a functional language, with object-oriented capabilities taking a back seat. Nonetheless, these kinds of patterns can be very useful.

11.6 Glossary

Chapter 12

Hashtables

A **hashtable** is like an array, but more general. In a list, the indices have to be integers; in a hashtable they can be (almost) any type. Hashtable functions are in the Hashtbl module - a great example of how computer scientists are lazy and don't want to write out more letters than absolutely necessary.

You can think of a hashtable as a mapping between a set of indices (which are called **keys**) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

As an example, we'll build a hashtable that maps from English to French words, so the keys and the values are all strings.

The function Hashtbl.create creates a new hashtable with no items. It takes one argument, an integer, that is the initial size of the table. The hashtable will grow as necessary, but we have to start with something. Adding elements above this initial guess is slower than merely changing the preallocated ``empty'' elements. However, it is always better to guess low and add more than preallocate too much. Think of this argument as a lower bound for your hashtable size.

```
# let eng2fr = Hashtbl.create 10;;
val eng2fr ('_a','_b') Hashtbl.t = <abstr>
```

Note that the type of the hashtable is Hashtbl.t. To add an element, use the add function:

```
# Hashtbl.add eng2fr "hello" "bonjour";;
- : unit = ()
```

Note that hashtables are mutable - we didn't create a new table and assign it to a variable, we changed the old value, and the function returned a unit.

To view an element, we use the find function.

```
# Hashtbl.find "hello";;
- : string = "bonjour"
```

www.dbooks.org

We can also replace or remove elements:

```
# Hashtbl.replace eng2fr "hello" "salut";;
- : unit = ()
# Hashtbl.find eng2fr "hello";;
- : string = "salut"
# Hashtbl.remove eng2fr "hello";;
- : unit = ()
# Hashtbl.find eng2fr "hello";;
Exception: not_found
```

You don't need to make keys unique. For example, we can have multiple translations for ``hello'':

```
# Hashtbl.add eng2fr "hello" "bonjour";;
# Hashtbl.find eng2fr "hello";;
 - : string = "bonjour"
# Hashtbl.add eng2fr "hello" "salut";;
# Hashtbl.find eng2fr "hello";;
 - : string = "salut"
# Hashtbl.find_all eng2fr "hello";;
 - : string list = ["bonjour"; "salut"]
```

When you add twice, the most recent add is what will be used, unless you use a function that specifically works with all the values, such as find_all, which returns a list of all values. replace and remove, however, only work on the most recent value - so if you remove a second value, you'll now be working with the first.

The length function returns the number of mappings (not necessarily the number of keys):

```
# Hashtbl.length eng2fr
- : int = 2
```

To check whether or not something is a key in a hashtable, use the mem function:

```
# Hashtbl.mem eng2fr "hello";;
- : bool = true
```

When finding items in a list, the amount of time it took to find an element grew proportionally to the length of that list. For remarkable reasons I won't explain, this isn't true for hashtables - it takes about the same amount of time to find a given element no longer how large the hashtable is!

Exercise 12.1 Write a function that reads the words in `words.txt` and stores them as keys in a hashtable. It doesn't matter what the values are. Then you can use the `mem` function as a fast way to check whether a string is in the hashtable.

If you did Exercise 8.11, you can compare the speed of this implementation with the list equivalent.

12.1 Hashtable as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using pattern matching.
2. You could create a list with 26 elements. Then you could convert each character to a number, use the number as an index into the list, and increment the appropriate counter.
3. You could create a hashtable with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
#load "str.cma";;

let histogram s =
let wordlist = Str.split (Str.regexp "\\b ") s in
let hist = Hashtbl.create 1 in
let addfunct e =
if (Hashtbl.mem hist e)
then let cntr = (Hashtbl.find hist e + 1)
in Hashtbl.replace hist e cntr
else Hashtbl.add hist e 1
in
List.iter addfunct wordlist;
hist
;;
```

This function takes a string, splits it into words, and then creates a hashtable with the words as keys and the number of times that word appeared as values. The name of the function is **histogram**, which is a statistical term for a set of counters (or frequencies).

The first line of the function creates an empty Hashtable, with assumed length of 1. If you know more information about the string you're going to pass to histogram, you might pass a more intelligent guess. It then iterates over the list, for each element either adding the previously-unseen element to the hashtable, or else increasing the counter for that element by one.

www.dbooks.org

Here's how it works:

```
let histprint tbl =
Hashtbl.iter (fun key value ->
print_string (key^":"^(string_of_int value)^"\n"))
tbl;;

let hist = histogram "hello day, the sun says hello";;

# histprint hist;;
sun:1
the:1
says:1
hello:2
day,:1
- : unit = ()
```

The histogram indicates that ``sun'', ``the'', and ``says'' all appear once, and hello appears twice. Note that it doesn't really handle punctuation the way we'd probably want it to, so the word ``day,'' appears once, not just the word ``day''.

Exercise 12.2 Rewrite `histogram` to better handle punctuation. If you did the exercises in chapter 9, this should be easy for you.

12.2 Iteration and Hashtables

We just saw an instance of using iteration over hashtables. Let's break it down a little more:

```
let histprint tbl =
Hashtbl.iter (fun key value ->
print_string (key^":"^(string_of_int value)^"\n"))
tbl;;

# histprint hist;;
sun:1
the:1
says:1
hello:2
day,:1
- : unit = ()
```

Iteration in hashtables works not unlike that of lists and arrays. The only real difference is that the function argument takes two arguments - first the key and then the value.

Exercise 12.3 Modify `print_hist` to print the keys and their values in alphabetical order.

12.3 Folding and Hashtables

You can also fold hashtables, which, again, is kind of what you would expect. Let's say we wanted to add up the number of words in our histogram that begin with the letter s:

```
let addSes tbl =
Hashtbl.fold (fun key value acc ->
if key.[0] = 's'
then acc+value
else acc)
tbl 0 ;;
```

One thing to note, there is no fold_left or fold_right for hashtables. This is because the mappings in the hashtable are not stored in a particular order.

12.4 Reverse lookup

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup**.

But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search.

Here is a function that takes a value and returns a reference list of keys :

```
let reverse_lookup tbl v =
let lst = ref [] in
Hashtbl.iter
(fun key value ->
if value==v
then lst := key::!lst )
tbl;
lst;;
```

Here is an example of a successful reverse lookup:

```
# let hist = histogram "hello day the sun says hello";;
# reverse_lookup hist 1;;
- : string list ref = {contents = "says"; "day"; "the"; "sun"}
```

Note again how they are out of order.

And a couple unsuccessful ones:

```
# reverse_lookup hist 3;;
- : string list ref = {contents = []}
# reverse_lookup hist 'a';;
Error: This expression has type char but an expression was expected of type int
```

www.dbooks.org

In some situations, this is probably the behavior you want: an error if the lookup is impossible, and an empty list if there are no keys. However, it's also possible that you'd want to raise an error if a key isn't found, rather than letting the program go on with an empty list. For this, there's the exception and raise functions:

```
exception UnfoundValue;;

let reverse_lookup tbl v =
let lst = ref [] in
Hashtbl.iter
(fun key value ->
if value==v
then lst := key::!lst )
tbl;
match !lst with
[] -> raise UnfoundValue
| h::t -> ();
!lst;;
```

exception defines a cnew exception. Note that the argument isn't exactly a string - it is not between quotes. This is actually a new type we haven't seen before, namely that of an exception. We can then raise this exception with the raise function.

Now, when we run the search with a value that doesn't exist, it will throw an exception error:

```
# reverse_lookup hist 3;;
Exception: UnfoundValue
```

You should always try to name exceptions such that they will make sense to your user. Remember that whoever else works with your code can't read your brain. Be nice to them.

In general, a reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Exercise 12.4 Modify `reverse_lookup` so that it returns only the first key that points to the value, then so it points to the greatest-valued (in the built-in `compare` function sense) key.

12.5 Memos

If you wrote and played with the fibonacci function from Exercise 5.7, you probably wrote a function that looks something like this:

```
let rec fibonacci n = match n with
x when x=0 -> 0
| x when x=1 -> 1
| x -> (fibonacci (n-1)) + (fibonacci (n-2));;
```

and you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. This is because the number of function calls required grows exponentially with each call.

In particular, consider how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**[1]. Here is an implementation of fibonacci using memos:

```
let known = Hashtbl.create 2;;
Hashtbl.add known 0 1;;
Hashtbl.add known 1 1;;
let rec fibonacci n =
if Hashtbl.mem known n
then Hashtbl.find known n
else let fn = (fibonacci (n-1)) + (fibonacci (n-2)) in
Hashtbl.add known n fn;
fn;;

let printknown () =
let n = ref 0 in
while Hashtbl.mem known !n do
print_string ("f("^(string_of_int !n)^"):"^
(string_of_int (Hashtbl.find known !n)));
print_newline ();
n := !n+1
done;;
```

known is a hashtable that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the hashtable, and return it. Thus, we build up a record of all the fibonacci numbers up to a highest call. We can view our record by calling printknown:

```
# fibonacci 10;;
- : int = 89http://en.literateprograms.org/Fibonacci_numbers_(OCaml)#Arbitrary_Precision
# printknown();;
f(0):0
f(1):1
f(2):2
f(3):3
f(4):5
f(5):8
```

[1] See wikipedia.org/wiki/Memoization.

www.dbooks.org

```
f(6):13
f(7):21
f(8):34
f(9):55
f(10):89
```

Exercise 12.5 Run this version of `fibonacci` and the original with a range of parameters and compare their run times.

In the previous example, known is created outside the function, so it belongs to the special frame that is the highest scope. Variables in this scope are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next. They are an example of using scope to your advantage, but you must always be careful when using scope.

12.6 Long integers

If I compute fibonacci(50), I get:

```
>>> fibonacci(50)
- : int=1037658242
```

which isn't too suspicious. It's the wrong answer, but it gets worse. Let's look at the record and see:

```
...
f(48):-811192543
f(49):-298632863
f(50):1037658242
```

Well, that doesn't look good at all. the 48-th and 49-th term are clearly not right, and they also clearly don't add up to f(50). Values with type int have a limited range, which we've exceeded. There are ways around this problem, but I won't bother with them here. If you want to find out more, I recommend looking at http://en.literateprograms.org/Fibonacci_numbers_(OCaml)#Arbitrary_Precision.

Exercise 12.6 Exponentiation of large integers is the basis of common algorithms for public-key encryption. Read the Wikipedia page on the RSA algorithm[2] and write functions to encode and decode messages.

12.7 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

[2] wikipedia.org/wiki/RSA.

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or modify the program so it reads only the first n lines.

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a ``sanity check'' because it detects results that are ``insane.''

Another kind of check compares the results of two different computations to see if they are consistent. This is called a ``consistency check.''

Pretty print the output: Formatting debugging output can make it easier to spot an error. Writing (or, better yet, finding built-in or distributed) functions to print things in human-readable ways can make your life a lot easier.

Again, time you spend building scaffolding can reduce the time you spend debugging.

12.8 Glossary

hashtable: A mapping from a set of keys to their corresponding values.

key-value pair: The representation of the mapping from a key to a value.

item: Another name for a key-value pair.

key: An object that appears in a hashtable as the first part of a key-value pair.

value: An object that appears in a hashtable as the second part of a key-value pair. This is more specific than our previous use of the word ``value.''

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python hashtables.

hash function: A function used by a hashtable to compute the location for a key.

www.dbooks.org

lookup: A hashtable operation that takes a key and finds the corresponding value.

reverse lookup: A hashtable operation that takes a value and finds one or more keys that map to it.

histogram: A set of counters.

memo: A computed value stored to avoid unnecessary future computation.

global variable: A variable defined outside a function. Global variables can be accessed from any function.

flag: A boolean variable used to indicate whether a condition is true.

declaration: A statement like global that tells the interpreter something about a variable.

12.9 Exercises

Exercise 12.7 If you did Exercise 8.9, you already have a function named `has_duplicates` that takes a list as a parameter and returns `True` if there is any object that appears more than once in the list.

Use a dictionary to write a faster, simpler version of `has_duplicates`.

Exercise 12.8 Two words are "rotate pairs" if you can rotate one of them and get the other (see `rotate_word` in Exercise 7.7).

Write a program that reads a wordlist and finds all the rotate pairs.

Exercise 12.9 Here's another Puzzler from *Car Talk*[3]:

> This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?
>
> Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack.' W-R-A-C-K, you know like to 'wrack with pain.' If I remove the first letter, I am left with a four-letter word, 'R-A-C-K.' As in, 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' It's a perfect homophone. If you put the 'w' back, and remove the 'r,' instead, you're left with the word, 'wack,' which is a real word, it's just not a homophone of the other two words.
>
> But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?

[3] `www.cartalk.com/content/puzzler/transcripts/200717`.

You can use the dictionary from Exercise 12.1 to check whether a string is in the word list.

To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from `www.speech.cs.cmu.edu/cgi-bin/cmudict`.

Write a program that lists all the words that solve the Puzzler.

www.dbooks.org

Chapter 13

Tuples

A tuple is an immutable sequence of values, not unlike lists. The important difference is that tuples can contain objects of any type.

13.1 Creating and Using Tuples

Syntactically, a tuple is a comma-separated list of values:

```
# let t = 'a', 'b', 'c', 'd', 'e';;
val t : char * char * char * char * char = ('a', 'b', 'c', 'd', 'e')
```

Note that the ocaml interpreter prints the type of a tuple as a series of types connected by asterisks. Although it is not necessary, it is common to enclose tuples in parentheses:

```
# let t = ('a', 'b', 'c', 'd', 'e');;
```

If you're passing a tuple as an argument, however, it does need to be enclosed in parentheses to tell the function how to group it.

Much like lists, there's no built-in way to access the elements of a tuple in general. If your tuple is a pair, you can use the fst and snd functions to access the first and second elements, respecitvely, but for larger tuples you'll need to write your own functions, or use arrays instead. Unlike lists, there isn't the idea of a tuple ``head'' and ``tail,'' either. In general, you need to use pattern matching to select elements of a tuple:

```
# let third_of_three trip = match trip with (_,_,x) -> x;;
val third_of_three 'a * 'b * 'c -> 'c = <fun>
# third_of_three (1,2,3);;
- : int = 3
```

13.2 Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a

www.dbooks.org

and b:

```
# let temp = a;;
# let a = b;;
# let b = temp;;
```

This solution is quite combersome, and wastes precious memory. This can more cleanly be accomplished with an and:

```
#let a = b and b = a;;
```

but that's a little cumbersome, too. This can be made much cleaner using **tuple assignment**:

```
# let a, b = b, a;;
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
# let a, b = 1, 2, 3;;
Error: This expression has type int * int * int
       but an expression was expected of type 'a * 'b
```

13.3 Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it makes sense to do it in one fell swoop:

```
# let divmod a b = a/b, a mod b;;
# divmod 5 2;;
- : int * int = (2, 1)
```

Or use tuple assignment to store the elements separately:

```
# let quot, rem = divmod(5, 2)
val q : int = 2
val r : int = 1
`
```

13.4 Lists and tuples

List.combine is a built-in function that takes two lists and combines them into a list of tuples where each tuple contains one element from each sequence.

This example combines a list of ints and a list of chars:

```
# let c = ['a'; 'b'; 'c'];;
# let t = [0, 1, 2];;
# List.combine t c;;
- : (int * char) list = [(1, 'a') (2, 'b') (3, 'c')]
```

The result is a list of tuples where each tuple contains an integer and a char.

You can use tuple assignment and recursion to traverse a list of tuples:

```
let print_tuple tup =
let t, c = tup in
print_string "(";
print_int t;
print_string ", ";
print_char c;
print_string")";
print_newline ();;

let rec tuplister tlst = match tlst with
[] -> ()
| h::t -> print_tuple h;
tuplister t;;

# let tlist = [(1, 'a'); (2, 'b'); (3, 'c')];;
# tuplister tlist;;
(1, 'a')
(2, 'b')
(3, 'c')
- : unit = ()
```

Exercise 13.1 Most of these methods have equivalent or similar functions for Arrays. You should look them up in the Array module and play with them, too.

13.5 Comparing tuples

The relational operators work with tuples; OCaml starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
# (0, 1, 2) < (0, 3, 4);;
- : bool = true
# (0, 1, 2000000) < (0, 3, 4);;
- : bool = true
```

13.6 Debugging

13.7 Glossary

www.dbooks.org

Chapter 14

Records and Custom Data Structures

Records are very similar to tuples in that they're ordered sets of information of any type. Records, however, are labelled - each piece of information gets a title. They are also custom-defined types: you have to tell OCaml ahead of time what the record is going to look like. You do this by using the type keyword:

```
# type identity = {
  firstName: string;
  lastName: string;
  sex: char;
  age: int
};;
type identity = {
  firstName : string;
  lastName : string;
  sex : char;
  age : int;
}
```

We can now create a record of the type identity like so:

```
let nmonje = {firstName = "Nicholas"; lastName = "Monje"; sex = 'm'; age = 21}
```

14.1 Tuples

We can access elements of a record using dot syntax:

```
# nmonje.age;;
- : int = 21
```

Record comparison works the same way as tuple comparison - it compares elements in order and returns a value based on the first different element:

```
type pair = {x: int; y: int};;
let a = {x = 1; y = 2};;
```

www.dbooks.org

```
let b = {x = 2; y = 2};;
let c = {x = 1; y = 3};;
let d = {x = 1; y = 2};;
# a<b
- : bool = true
# a<c
- : bool = true
# b<c
- : bool = false
# a = d
- : bool = true
# a == d
- : bool = false
```

Wait, what? a and d are ``='' but not ``==''? What does that mean?

The single equals sign tests for what's called **structural** or **deep equality**, or tests to see if two objects' contents are the same. The double equals sign tests for **physical** or **shallow equality**, or whether or not two objects reference the same thing. In most of what we've seen so far, the two are equivalent. If they simply point to integers, then the two equalities effectively mean the same thing. However, now that we're looking at more ``object-y'' objects, there can be a subtle difference. For example, two references are not necessarily shallowly equivalent:

```
# let x = ref 10;;
# let y = ref 10;;
# x = y;;
- : bool = true
# x == y;;
- : bool = false
```

x and y have the same value, but they point to two different spots in the computer's memory, so they are not shallowly equivalent. It's perhaps a little bit weird that two things are deeply equivalent but not shallowly equivalent, but just think of it like people: we're different on the outside, but deep down we're all the same. Or something like that.

14.2 Enumerated Types

The type keyword isn't just for records. It can be used to define many different kinds of types. One important custom type is called an **enumerated type**, which has a discrete, finite set of values:

```
# type enum = A | B | C';
type enum = A | B | C
# let a = A;;
val a : enum = A
# let d = D;;
Error: Unbound constructor D
```

The newly-defined enum type has three possible values - A, B, or C. This example isn't particularly interesting, but there isn't a whole lot built-in around enumerated types - or any enumerated type, really. They're entirely custom-defined, and you must write code to interact with them.

There is implicit comparison, though, determined by the order in which you defined the type:

```
# A < B;;
- : bool = true
# C > B;;
- : bool = true
# A > C;;
- : bool = false
```

14.3 Aggregate types

Next, you can define your own **aggregate types**, which are types that are composed of basic types:

```
# type agg = Agg of int * string;;
# let a = Agg (1, "hi");;
val a : agg = Agg (1, "hi")
```

Aggregate types can be enumerated, and can be recursive. Here's a basic definition of a tree of integers:

```
type tree = Tree of int * tree | Leaf of int
```

which would allow us to define the following tree:

```
let t = Tree (3, Tree( 2, Leaf 1));;
```

This tree is, of course, not very interesting because it can't branch. We can quite easily create a binary tree:

```
type bitree = Tree of int * tree * tree | Leaf of int | Null;;
```

The Null allows us to make this tree not full - if we don't want to branch in both directions, we could make one direction be Null. We could furhter expand this definition to any determined finite size tree. However, if we want to leave the number of nodes ambiguous, we could use a list of trees:

```
type mtree = Tree of int * mtree list | Leaf of int | Null;;
let mt = Tree (3, [Tree (2, [Null]); Tree (2, [Tree (1, [Leaf 1])])]);;
```

Which of course is kind of nasty because *every* node must be a list, even if it contains only one element, which you could easily get around by including another enumeration for Tree of int* mtree. Even then, this is where parentheses matching can get very important. You might want to consider finding a development environment that will help you with that.

Of course, all these enumerated types only work on integers. We can also define our tree to be polymorphic. I'm going to stick to a binary tree for this example:

www.dbooks.org

```
type 'a ptree = PTree of 'a * 'a ptree * 'a ptree | PLeaf of 'a | PNull;;

let pt = PTree ("Root", PTree ("Node", PLeaf "Leaf", PLeaf "Leaf"), PLeaf "Leaf");;
```

Of course, in this example, all the data must be of the same type.

Exercise 14.1 Rewrite the polymorphic tree example above such that it can store data of two different types. Note that different polymorphic types are written as `'a`, `'b`, and so on.

14.4 Debugging

One thing to be careful of is redefining types. For example:

14.5 Glossary

tuple: An immutable sequence of elements which may be of different types.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

gather: The operation of assembling a variable-length argument tuple.

scatter: The operation of treating a sequence as a list of arguments.

DSU: Abbreviation of ``decorate-sort-undecorate,'' a pattern that involves building a list of tuples, sorting, and extracting part of the result.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape (of a data structure): A summary of the type, size and composition of a data structure.

14.6 Exercises

Exercise 14.2 Write a function called `most_frequent` that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at `wikipedia.org/wiki/Letter_frequencies`.

Exercise 14.3 More anagrams!

1. Write a program that reads a word list from a file (see Section **??**) and prints all the sets of words that are anagrams.

   Here is an example of what the output might look like:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

Hint: you might want to build a dictionary that maps from a set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?

2. Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, and so on.

3. In Scrabble a "bingo" is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What set of 8 letters forms the most possible bingos? Hint: there are seven.

4. Two words form a "metathesis pair" if you can transform one into the other by swapping two letters[1]; for example, "converse" and "conserve." Write a program that finds all of the metathesis pairs in the dictionary. Hint: don't test all pairs of words, and don't test all possible swaps.

   You can download a solution from `thinkpython.com/code/anagram_sets.py`.

Exercise 14.4 Here's another Car Talk Puzzler[2]:

> What is the longest English word, that remains a valid English word, as you remove its letters one at a time?
>
> Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?
>
> I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most, so here are some suggestions:

1. You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.

2. Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.

3. The wordlist I provided, `words.txt`, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.

4. To improve the performance of your program, you might want to memoize the words that are known to be reducible.

You can see my solution at `thinkpython.com/code/reducible.py`.

[1] This exercise is inspired by an example at `puzzlers.org`.

[2] `www.cartalk.com/content/puzzler/transcripts/200651`.

www.dbooks.org

Chapter 15

Putting the O in Ocaml Part 2: Objects and Classes

In general, OCaml programming (and programmers) prefer function-oriented programming, and you generally can - and should - use primarily functional solutions to programming. However, object-oriented programming can be very useful in certain situations.

Object-oriented programming with objects would be just oriented programming, and that's not very fun. Objects can take many forms, so the closest thing we can get to a definition is an object is something a variable points to that has a type and value, and possibly methods. What this means will make more sense with an example.

Go to inserturlhere and download oturtle.zip. Unzip the files using your favorite archive software.[1]

Now, open the interpreter in the same directory and type #use "oturtle.ml". You can now call functions defined within the Oturtle module. You'll better understand what's going on here later. Note that this loads the very useful Graphics module as well. You should briefly check out this module online.

What Oturtle does is facilitates a particular method of drawing. Namely, we have turtles, which can draw along their path.[2] We can control the turtle by telling it to move, and draw as it goes along (we can also tell it to move *without* drawing).

15.1 Turtles

Now that we conceptually understand what Oturtle does, open a blank figure with Oturtle.init ().

We can now create a turtle like so:

[1] I recommend 7zip for Windows.
[2] The reason it's a turtle is of course whimsical.

www.dbooks.org

```
# let bob = new Oturtle.turtle;;
```

This is a syntax we haven't seen before. We've created a variable, bob which refers to an object, namely a turtle. The keyword new tells us to create a new object of the specified type (without it, bob would point to the type, or **class** itself). The new turtle is of a random color and exists

Now, let's see what Bob looks like. In order to draw Bob, we can call the draw **method**. A method is a like a function, but it is written into the definition of an object and it is kind of like the object is doing something. The syntax is quite unlike what we've seen before:

```
# bob#draw;;
```

And Bob shows up in the middle of the canvas! Isn't he cute? Note that this method does not take an argument - not even the unit argument, like all functions we've seen. A method ***can*** take an argument, and we'll see that in a second, but it's not necessary. There is currently no method of erasing bob, so he'll always show up right there, even after we've moved him someplace else. This functionality will hopefully be implemented in a later version.

Now, that's adorable, but Bob wouldn't be a very good turtle if he couldn't move. Perhaps unsurprisingly, telling Bob to move is also a method. In fact, there are two methods: step, which moves a single pixel, and advance, which takes an integer argument and moves Bob that many pixels.

```
# bob#step;;
# bob#draw;;
# bob#advance 100;;
# bob#draw;;
```

You'll notice that a pixel isn't very far - he barely looks like he's moved at all!

You can also tell Bob to turn:

```
# bob#turn 90;;
# bob#advance 30;;
# bob#draw;;
```

turn also takes an integer argument and rotates Bob that many degrees. If you're not familiar with degrees of a circle, you should read about them on wikipedia.

We can tell Bob to start drawing by lowering his pen, or stop drawing by telling him to raise his pen:

```
bob#lower_pen;;
bob#advance 100;;
bob#draw;;
bob#raise_pen;;
bob#advance 30;;
bob#draw;;
```

When the pen is lowered, the turtle draws a line along his path. When the pen is raised, it doesn't.

Besides, methods, objects also have values. Values preserve the state of the object - without values, an object doesn't contain any information. Values can be either mutable or immutable, depending on whether you set them to be or not (values are by default immuatble. A turtle, for example, has mutable values x and y coordinates x and y, direction theta, all integers as well as a an immutable color color, which is a special type Graphics.color, as well as pen status pen, which is a custom enumerated type with values Raised and Lowered.

In general, OCaml doesn't provide a way to acces values of an object. Methods, as we have already seen, can change the values of mutable values - advance, step, and turn all change Bob's values. There are also often methods which return values:

```
# bob#get_pos;;
- : int * int = (400, 254)
# bob#get_dir;;
- : int = 90
```

These functions don't change values (though a method can both change and return a value), but they do read and return values.

Exercise 15.1

15.2 Defining Modules and Classes

Now, let's lift the hood on this code a little bit. Open oturtle.ml in your favorite text editor. You'll notice quite a few constructions we haven't seen before. First, note how we define a module. The syntax module Oturtle = is much like a let binding, but for modules. The functions and types defined by the class are enclosed within the struct... end construction. You can include values within a module as well.

Defining a class operates in much the same way. The class turtle = is just like a let-binding except for classes.[3] We can also have the turtle take arguments to initialize it. Consider the following alternative definition of a turtle:

```
class turtle xi yi = object (self)
  val mutable x = xi
  val mutable y = yi
  ...
end
```

Now, to create the object, we would call:

```
let bob = new Oturtle.turtle 0 0;;
```

[3]Note that classes do not have to be defined within modules.

www.dbooks.org

which will create bob in the bottom-left corner of the screen.

Within class definitions, all our values and methods are contained within the object... end construction. The (self) lets us call the object within it's definition, just like we would call it outside of the class definition. For example, consider the definition of step, which calls self#advance. In general, we could call it anything, not just self, but ``self'' is very common, and it's not generally good form to call it something else unless it really makes sense to do so.

15.3 The `Str` module

There's one more module you might want to be familiar with, that's quite common in OCaml coding. It's the Str module, and it allows for a lot of functionality that object-oriented programmers would expect to deal with strings. To define methods or values, we use the method or val keywords, again, just like let-bindings except for methods and values.

Note that, within both method and class definitions, we do not use semicolons to denot the end of a definition; methods and classes are all defined within a single ``line'' of code.

15.4 Debugging

Because class definitions do not use semicolons to denote the end of lines, figuring out how to define methods that are more than very simple statements can get a little complicated. In general, you can always enclose code in more parentheses to help the compiler figure it out, especially in if or match clauses.

While it's generally nasty to include more than necessary, using the absolute minimum isn't necessarily recommended either. You want to use exactly the number necessary to make the code clear to both reader and compiler. Sometimes you do have to give up a little readability so that it will compile, but that's often[4] a sign that your code is messy to begin with.

15.5 Glossary

Words

15.6 Exercises

Exercises

[4] not always!

Chapter 16

Case study: data structure selection

16.1 Word frequency analysis

As usual, you should at least attempt the following exercises before you read my solutions.

Exercise 16.1 Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.

Hint: The `Str` module will probably proove useful here.

Exercise 16.2 Go to Project Gutenberg (`gutenberg.net`) and download your favorite out-of-copyright book in plain text format.

Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.

Then modify the program to count the total number of words in the book, and the number of times each word is used.

Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

Exercise 16.3 Modify the program from the previous exercise to print the 20 most frequently-used words in the book.

Exercise 16.4 Modify the previous program to read a word list (see Section **??**) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that *should* be in the word list, and how many of them are really obscure?

16.2 Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For

www.dbooks.org

some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The Random module provides functions that generate pseudorandom numbers (which I will simply call ``random'' from here on).

To begin, you must initialize the random number generator. To do so, call Random.self_init. It takes a unit argument and returns a unit. There are other methods of initialization, but this is arguably the ``most random.'' (The other methods take ``seeds'' and will return the same sequence every time for a given seed.)

We can then return random integers, floots, booleans, or other number types we haven't covered. With the exception of booleans, all of them take one argument, or the same type it will return, as an upper bound. Each function will return a random number between 0 and the bound. Conveniently, each of the functions is named by type.

```
# Random.int 10;;
- : int = 5
# Random.float 1.0;;
- : float = 0.438748135348833446
# Random.bool ();;
- : bool = false;;
```

Exercise 16.5 Write a function named `choose_from_hist` that takes a histogram as defined in Section 12.1 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
# let t = ["a"; "a"; "b"]
# let h = histogram(t);
>>> print h
{'a': 2, 'b': 1}
```

your function should `'a'` with probability $2/3$ and `'b'` with probability $1/3$.

16.3 Word histogram

Here is a program that reads a file and builds a histogram of the words in the file:

```
import string

def process_file(filename):
```

```
    h = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, h)
    return h

def process_line(line, h):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()

        h[word] = h.get(word, 0) + 1

hist = process_file('emma.txt')
```

This program reads emma.txt, which contains the text of *Emma* by Jane Austen.

process_file loops through the lines of the file, passing them one at a time to process_line. The histogram h is being used as an accumulator.

process_line uses the string method replace to replace hyphens with spaces before using split to break the line into a list of strings. It traverses the list of words and uses strip and lower to remove punctuation and convert to lower case. (It is a shorthand to say that strings are ``converted;'' remember that string are immutable, so methods like strip and lower return new strings.)

Finally, process_line updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```
def total_words(h):
    return sum(h.values())
```

The number of different words is just the number of items in the dictionary:

```
def different_words(h):
    return len(h)
```

Here is some code to print the results:

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

And the results:

```
Total number of words: 161073
Number of different words: 7212
```

www.dbooks.org

16.4 Most common words

To find the most common words, we can apply the DSU pattern; most_common takes a histogram and returns a list of word-frequency tuples, sorted in reverse order by frequency:

```
def most_common(h):
    t = []
    for key, value in h.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

Here is a loop that prints the ten most common words:

```
t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
    print word, '\t', freq
```

And here are the results from *Emma*:

```
The most common words are:
to      5242
the     5204
and     4897
of      4293
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

16.5 Optional parameters

We have seen built-in functions and methods that take a variable number of arguments. It is possible to write user-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram

```
def print_most_common(hist, num=10)
    t = most_common(hist)
    print 'The most common words are:'
    for freq, word in t[0:num]:
        print word, '\t', freq
```

The first parameter is required; the second is optional. The **default value** of num is 10.

If you only provide one argument:

```
print_most_common(hist)
```

num gets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

num gets the value of the argument instead. In other words, the optional argument **overrides** the default value.

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

16.6 Dictionary subtraction

Finding the words from the book that are not in the word list from words.txt is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in another set (the words in the list).

subtract takes dictionaries d1 and d2 and returns a new dictionary that contains all the keys from d1 that are not in d2. Since we don't really care about the values, we set them all to None.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in words.txt, we can use process_file to build a histogram for words.txt, and then subtract:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

Here are some of the results from *Emma*:

```
The words in the book that aren't in the word list are:
 rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Some of these words are names and possessives. Others, like ``rencontre,'' are no longer in common use. But a few are common words that should really be in the list!

Exercise 16.6 Python provides a data structure called `set` that provides many common set operations. Read the documentation at `docs.python.org/lib/types-set.html` and write a program that uses set subtraction to find words in the book that are not in the word list.

www.dbooks.org

16.7 Random words

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

The expression [word] * freq creates a list with freq copies of the string word. The extend method is similar to append except that the argument is a sequence.

Exercise 16.7 This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big.

An alternative is:

1. Use `keys` to get a list of the words in the book.
2. Build a list that contains the cumulative sum of the word frequencies (see Exercise **??**). The last item in this list is the total number of words in the book, *n*.
3. Choose a random number from 1 to *n*. Use a bisection search (See Exercise 8.11) to find the index where the random number would be inserted in the cumulative sum.
4. Use the index to find the corresponding word in the word list.

Write a program that uses this algorithm to choose a random word from the book.

16.8 Markov analysis

If you choose words from the book at random, you can get a sense of the vocabulary, you probably won't get a sentence:

```
this the small regard harriet which knightley's it most things
```

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like ''the'' to be followed by an adjective or a noun, and probably not a verb or adverb.

One way to measure these kinds of relationships is Markov analysis[1], which characterizes, for a given sequence of words, the probability of the word that comes next. For example, the song *Eric, the Half a Bee* begins:

[1] This case study is based on an example from Kernighan and Pike, *The Practice of Programming*, 1999.

```
Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity.  D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?
```

In this text, the phrase ``half the'' is always followed by the word ``bee,'' but the phrase ``the bee'' might be followed by either ``has'' or ``is''.

The result of Markov analysis is a mapping from each prefix (like ``half the'' and ``the bee'') to all possible suffixes (like ``has'' and ``is'').

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

For example, if you start with the prefix ``Half a,'' then the next word has to be ``bee,'' because the prefix only appears once in the text. The next prefix is ``a bee,'' so the next suffix might be ``philosophically,'' ``be'' or ``due.''

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length. The length of the prefix is called the ``order'' of the analysis.

Exercise 16.8 Markov analysis:

1. Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length two, but you should write the program in a way that makes it easy to try other lengths.

2. Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from *Emma* with prefix length 2:

   > He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.

   For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

   What happens if you increase the prefix length? Does the random text make more sense?

3. Once your program is working, you might want to try a mash-up: if you analyze text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.

www.dbooks.org

16.9 Data structures

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

- How to represent the prefixes.
- How to represent the collection of possible suffixes.
- How to represent the mapping from each prefix to the collection of possible suffixes.

Ok, the last one is the easy; the only mapping type we have seen is a dictionary, so it is the natural choice.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings. For the suffixes, one option is a list; another is a histogram (dictionary).

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is ``Half a,'' and the next word is ``bee,'' you need to be able to form the next prefix, ``a bee.''

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can't append or remove, but you can use the addition operator to form a new tuple:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

shift takes a tuple of words, prefix, and a string, word, and forms a new tuple that has all the words in prefix except the first, and word added to the end.

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see Exercise 16.7).

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is run time. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the in operator is faster for dictionaries than for lists, at least when the number of elements is large.

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This

approach is called **benchmarking**. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is no need to go on. If not, there are tools, like the profile module, that can identify the places in a program that take the most time.

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after run time.

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

16.10 Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

reading: Examine your code, read it back to yourself, and check that it says what you meant to say.

running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

ruminating: Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

retreating: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can starting rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall

www.dbooks.org

into a pattern I call ``random walk programming,'' which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

16.11 Glossary

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

pseudorandom: Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.

default value: The value given to an optional parameter if no argument is provided.

override: To replace a default value with an argument.

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

16.12 Exercises

Exercise 16.9 The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages[2]. Specifically, it predicts that the frequency, f, of the word with rank r is:

[2] See wikipedia.org/wiki/Zipf's_law.

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

$$\log f = \log c - s \log r$$

So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with $\log f$ and $\log r$. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s?

www.dbooks.org

Appendix A

Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a def statement yields the somewhat redundant message SyntaxError: invalid syntax.
- Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes the runtime error ‘‘maximum recursion depth exceeded.’’
- Semantic errors are problems with a program that runs without producing error messages but doesn’t do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

A.1 Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is

www.dbooks.org

not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
3. Make sure that any strings in the code have matching quotation marks.
4. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
5. An unclosed opening operator---(, {, or [---makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
6. Check for the classic = instead of == inside a conditional.
7. Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems. The best way to avoid this problem is to use a text editor that knows about Python and generates consistent indentation.

If nothing works, move on to the next section...

A.1.1 I keep making changes and it makes no difference.

If the interpreter says there is an error and you don't see it, that might be because you and the interpreter are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run.

If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run it again. If the interpreter doesn't find the new error, you are not running the new code.

There are a few likely culprits:

- You edited the file and forgot to save the changes before running it again. Some programming environments do this for you, but some don't.

- You changed the name of the file, but you are still running the old name.
- Something in your development environment is configured incorrectly.
- If you are writing a module and using import, make sure you don't give your module the same name as one of the standard Python modules.
- If you are using import to read a module, remember that you have to restart the interpreter or use reload to read a modified file. If you import the module again, it doesn't do anything.

If you get stuck and you can't figure out what is going on, one approach is to start again with a new program like ``Hello, World!,'' and make sure you can get a known program to run. Then gradually add the pieces of the original program to the new one.

A.2 Runtime errors

Once your program is syntactically correct, Python can compile it and at least start running it. What could possibly go wrong?

A.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the ``Flow of Execution'' section below.

A.2.2 My program hangs.

If a program stops and seems to be doing nothing, it is ``hanging.'' Often that means that it is caught in an infinite loop or infinite recursion.

- If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says ``entering the loop'' and another immediately after that says ``exiting the loop.''

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the ``Infinite Loop'' section below.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a ``RuntimeError: Maximum recursion depth exceeded'' error. If that happens, go to the ``Infinite Recursion'' section below.

  If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the ``Infinite Recursion'' section.

www.dbooks.org

- If neither of those steps works, start testing other loops and other recursive functions and methods.
- If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the ``Flow of Execution'' section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print  "x: ", x
    print  "y: ", y
    print  "condition: ", (x > 0 and y < 0)
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be false. If the loop keeps going, you will be able to see the values of x and y, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a Maximum recursion depth exceeded error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a print statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of Execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each function with a message like ``entering function foo,'' where foo is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

A.2.3 When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked *that*, and so on. In other words, it traces the sequence of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError: You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

TypeError: There are several possible causes:

- You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
- There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
- You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is self. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError: You are trying to access an element of a dictionary using a key that the dictionary does not contain.

AttributeError: You are trying to access an attribute or method that does not exist. Check the spelling! You can use dir to list the attributes that do exist.

If an AttributeError indicates that an object has NoneType, that means that it is None. One common cause is forgetting to return a value from a function; if you get to the end of a function without hitting a return statement, it returns None. Another common cause is using the result from a list method, like sort, that returns None.

IndexError: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a print statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

www.dbooks.org

The Python debugger (pdb) is useful for tracking down Exceptions because it allows you to examine the state of the program immediately before the error. You can read about pdb at docs.python.org/lib/module-pdb.html.

A.2.4 I added so many `print` statements I get inundated with output.

One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are searching a list, search a *small* list. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think shouldn't affect the program, and it does, that can tip you off.

A.3 Semantic errors

In some ways, semantic errors are the hardest to debug, because the interpreter provides no information about what is wrong. Only you know what the program is supposed to do.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed print statements is often short compared to setting up the debugger, inserting and removing breakpoints, and ''stepping'' the program to where the error is occurring.

A.3.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other OCaml modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Python, you might write:

www.dbooks.org

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
 y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence.

A.3.3 I've got a function or method that doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].removeMatches()
```

you could write:

```
count = self.hands[i].removeMatches()
return count
```

Now you have the opportunity to display the value of count before returning.

A.3.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these symptoms:

- Frustration and rage.
- Superstitious beliefs (''the computer hates me'') and magical thinking (''the program only works when I wear my hat backward'').
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

A.3.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you are prepared. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

www.dbooks.org

Index

abecedarian, 51
access, 56
accumulator
    histogram, 107
Ackerman function, 42
addition with carrying, 46
algorithm, 4, 9, 46, 110
    Euclid, 43
    RSA, 86
    square root, 47
ambiguity, 6
anagram, 61
anagram set, 98
and operator, 32
Anonymous functions, 59
argument, 21, 23, 25, 29
arithmetic operator, 13
assignment, 18
    tuple, 91--93, 98
assignment statement, 12
AttributeError, 121
Austin, Jane, 107

base case, 42
benchmarking, 113, 114
big, hairy expression, 123
binary search, 62
bingo, 99
birthday paradox, 61
bisection search, 62
bisection, debugging by, 47
body, 23, 29, 35
boolean expression, 31, 35
borrowing, subtraction with, 46
bracket operator, 49
branch, 35
bug, 4, 9

calculator, 19
caml.inria.fr, 10
Car Talk, 88, 99
carrying, addition with, 46
case-sensitivity, variable names, 17
chained conditional, 33
char type, 11
character, 18, 49
colon, 118
comment, 17, 19
comparison
    string, 52
    tuple, 93
compile, 1, 9
composition, 22, 25, 29
compound statement, 35
concatenation, 19, 26, 51
    list, 56
condition, 35, 120
conditional, 118
    chained, 33
    nested, 35
conditional execution, 32
conditional statement, 32, 35
cons operator, 55
consistency check, 87
counter, 53, 81
cummings, e. e., 4
Currying, 27

data structure, 98, 112
dead code, 122
debugger (pdb), 122
debugging, 4, 8, 9, 17, 28, 41, 53,
        61, 86, 98, 113, 117
    by bisection, 47
    emotional response, 8, 124
    experimental, 5
    superstition, 124
declaration, 88
default value, 108, 114
definition
    function, 22
    recursive, 99
deterministic, 105, 114

development plan
    incremental, 118
    random walk programming, 114, 124
diagram
    stack, 26
    state, 12
dictionary, 121
    lookup, 83
    looping with, 82
    reverse lookup, 83
    subtraction, 109
Directive, 2
divisibility, 31
documentation, 10
dot notation, 29
Doyle, Arthur Conan, 5
DSU pattern, 98, 108
duplicate, 61, 62, 88

element, 55, 61
emotional debugging, 8, 124
empty list, 55
empty string, 53
encapsulation, 46
encryption, 86
epsilon, 46
error
    compile-time, 117
    runtime, 4, 17, 39, 117
    semantic, 4, 12, 17, 117, 122
    syntax, 4, 17, 117
error message, 4, 8, 12, 17, 117
escape character, 7
Euclid's algorithm, 43
evaluate, 14
exception, 4, 9, 17, 117, 121
    AttributeError, 121
    IndexError, 50, 121
    KeyError, 121
    NameError, 121
    RuntimeError, 39
    SyntaxError, 22
    TypeError, 49, 121
    ValueError, 92
executable, 2, 9
experimental debugging, 5, 113
expression, 13, 14, 19
    big and hairy, 123
    boolean, 31, 35

Fermat's Last Theorem, 36
fibonacci function, 85
filter pattern, 61
find function, 51
flag, 88
float type, 11
floating-point, 18, 46
flow of execution, 24, 29, 120
For loop, 72
for loop, 58
formal language, 5, 9
format operator, 121
frame, 26
frequency, 81
    letter, 98
    word, 105, 114
frustration, 124
function, 22, 28
    ack, 42
    fibonacci, 85
    find, 51
    len, 29
    log, 21
    randint, 61
    recursive, 37
    reload, 119
    sqrt, 22
    String.length, 50
    zip, 92
function argument, 25
function call, 21, 29
function definition, 22, 24, 29
function frame, 26
function parameter, 25
function, math, 21
function, reasons for, 28
function, trigonometric, 21
function, tuple as return value, 92
Functional Programming, 7
Functions
    Anonymous, 59
    Currying, 27

gather, 98
GCD (greatest common divisor), 43
global variable, 88
greatest common divisor (GCD), 43
grid, 30
Guarded Patterns, 35

hanging, 119

www.dbooks.org

hash function, 87
hashtable, 79, 80, 87
header, 23, 29, 118
Hello, World, 7
high-level language, 1, 8
Higher-Order Functions, 25
histogram, 81, 88
    random choice, 106, 110
    word frequencies, 106
HOF, 25
Holmes, Sherlock, 5
homophone, 89

if statement, 32
immutability, 53
implementation, 81, 87, 112
in, 15
incremental development, 118
indentation, 118
index, 49, 53, 56, 79, 121
    starting at zero, 49
IndexError, 50, 121
infinite loop, 119, 120
infinite recursion, 39, 42, 119, 120
int type, 11
integer, 18
    long, 86
interactive mode, 2, 9
interlocking words, 62
interpret, 1, 9
invocation, 53
item, 53, 55
    hashtable, 87
item update, 58

key, 79, 87
key-value pair, 79, 87
keyboard input, 33
KeyError, 121
keyword, 13, 19, 118

language
    formal, 5
    high-level, 1
    low-level, 1
    natural, 5
    programming, 1
    safe, 4
len function, 29
let, 15
letter frequency, 98
letter rotation, 53, 88
Linux, 5
list, 55, 61
    concatenation, 56
    element, 56
    empty, 55
    nested, 55
    of tuples, 93
    operation, 56
    traversal, 58, 61
literalness, 6
local variable, 26, 29
log function, 21
logarithm, 115
logical operator, 31, 32
long integer, 86
lookup, 88
lookup, dictionary, 83
loop
    condition, 120
    for, 58
    infinite, 120
Looping, 72
looping
    with dictionaries, 82
low-level language, 1, 8

map pattern, 61
mapping, 111
Markov analysis, 110
mash-up, 111
math function, 21
McCloskey, Robert, 51
membership
    binary search, 62
    bisection search, 62
    hashtable, 80
    set, 80
memo, 85, 88
mental model, 123
metathesis, 99
method, 53
    string, 53
model, mental, 123
module, 7, 29
    pprint, 87
    profile, 113
    random, 61, 106
    reload, 119
    string, 105

modulus operator, 31, 35
multiline string, 118

NameError, 121
natural language, 5, 9
nested conditional, 33, 35
nested list, 55, 61
Newton's method, 45
not operator, 32
number, random, 105

object code, 2, 9
operand, 13, 19
operator, 13, 19
    and, 32
    bracket, 49
    cons, 55
    format, 121
    logical, 31, 32
    modulus, 31, 35
    not, 32
    or, 32
    overloading, 16
    relational, 32
    string, 16
operator, arithmetic, 13
optional parameter, 108
or operator, 32
order of operations, 16, 18, 123
override, 109, 114

palindrome, 42
parameter, 25, 26, 29
    optional, 108
parentheses
    empty, 23
    matching, 4
    overriding precedence, 16
    parameters in, 25
    tuples in, 91
parse, 6, 9
Partial Application, 27
pattern
    DSU, 108
    filter, 61
    map, 61
    reduce, 61
    search, 52, 53
    swap, 91
Pattern Matching, 34
Pattern-Matching
    Guarded, 35
pdb (Python debugger), 122
PEMDAS, 16
pi, 48
plain text, 105
poetry, 6
portability, 1, 8
pprint module, 87
precedence, 19, 123
precondition, 61
prefix, 111
pretty print, 87
print statement, 7, 9, 122
problem solving, 1, 8
profile module, 113
program, 3, 9
Programming
    Functional, 7
programming language, 1
Programming Paradigms, 7
    Functional, 7
    Object-Oriented, 7
Project Gutenberg, 105
prompt, 2, 9, 34
prose, 6
pseudorandom, 105, 114
Puzzler, 88, 99
Python debugger (pdb), 122

quotation mark, 7, 11, 118

radian, 21
rage, 124
Ramanujan, Srinivasa, 48
randint function, 61
random function, 106
random module, 61, 106
random number, 105
random text, 111
random walk programming, 114, 124
Read functions, 33
Recursion
    Tail-end, 40
recursion, 37, 42
    infinite, 39, 120
    traversal, 50
recursive definition, 99
reduce pattern, 61
reducible word, 89, 99
redundancy, 6

www.dbooks.org

References, 15, 71
relational operator, 32
reload function, 119
return statement, 124
return value, 21, 29
    tuple, 92
reverse lookup, dictionary, 83
reverse lookup, hashtable, 88
reverse word pair, 62
rotation
    letters, 88
rotation, letter, 53
RSA algorithm, 86
rules of precedence, 16, 19
running pace, 19
runtime error, 4, 17, 39, 117, 121
RuntimeError, 39

safe language, 4
sanity check, 87
scaffolding, 87
scatter, 98
Scope, 15
scope, 15
Scrabble, 99
script, 2, 9
script mode, 2, 9
search pattern, 52, 53
search, binary, 62
search, bisection, 62
semantic error, 4, 9, 12, 17, 117, 122
semantics, 4, 9
sequence, 49, 53, 55, 91
set, 109
    anagram, 98
set membership, 80
shape, 98
sine function, 21
slice, 53
source code, 2, 9
sqrt function, 22
square root, 45
stack diagram, 26
state diagram, 12, 18
statement, 18
    assignment, 12
    conditional, 32, 35
    for, 58
    if, 32
    print, 7, 9, 122
    return, 124
Strictly typed, 14
string, 11, 18
    comparison, 52
    multiline, 118
    operation, 16
string method, 53
String module, 50
string module, 105
string type, 11
String.length function, 50
structure, 6
subexpressions, 14
subtraction
    dictionary, 109
    with borrowing, 46
suffix, 111
superstitious debugging, 124
swap pattern, 91
syntax, 4, 9, 118
syntax error, 4, 9, 17, 117
SyntaxError, 22

Tail-end Recursion, 40
tail-end recursion, 42
temporary variable, 123
test case, minimal, 122
testing
    interactive mode, 2
    minimal test case, 122
text
    plain, 105
    random, 111
token, 6, 9
Toplevel, 2
toplevel, 9
traceback, 39, 41, 121
traversal, 50, 52, 53, 82, 93, 107
    list, 58
triangle, 36
trigonometric function, 21
tuple, 91, 92, 98
    as key in dictionary, 112
    assignment, 91
    comparison, 93
tuple assignment, 92, 93, 98
type, 11, 18
    char, 11
    float, 11

    hashtable, 79
    int, 11
    list, 55
    long, 86
    set, 109
    str, 11
    tuple, 91
    unit, 12, 23
TypeError, 49, 121
typographical error, 113

underscore character, 13
uniqueness, 62
unit type, 12, 18, 23
update
    histogram, 107
    item, 58
use before def, 17
User input, 33

value, 11, 18, 87
    default, 108
    tuple, 92
ValueError, 92
variable, 12, 18
    local, 26
    temporary, 123
Variables
    References, 15

While loop, 72
whitespace, 118
word frequency, 105, 114
word, reducible, 89, 99

zero, index starting at, 49
zip function, 92
Zipf's law, 114

www.dbooks.org