How to think like a computer scientist

Allen B. Downey
adapted to Pop11 by Waldek Hebisch

Pop11 Version, First Edition

How to think like a computer scientist
Pop11 Version, First Edition

This book was typeset by the author using LaTeX and dvips, which are both free, open-source programs.

Contents

Chapter 1

The way of the program

The goal of this book, and this class, is to teach you to think like a computer scientist. I like the way computer scientists think because they combine 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**. By that I mean 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, which is 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 What is a programming language?

The programming language you will be learning is Pop11. Pop11 is an example of a **high-level language**; other high-level languages you might have heard of are Pascal, C, Java and Lisp.

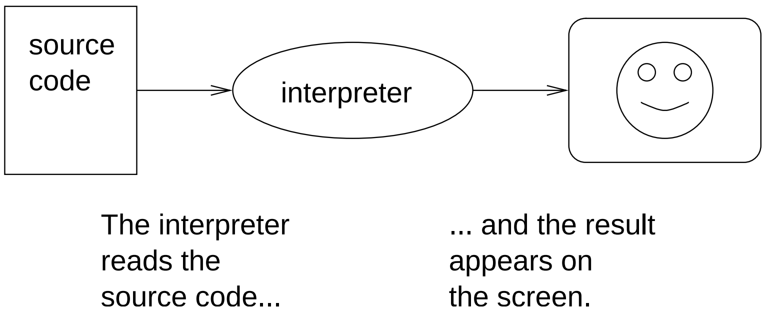
Pop11 is implemented by Poplog system. Beside Pop11 Poplog offers other languages: Prolog, Lisp and Standard ML. Poplog used to be a very expensive commercial product and only relatively recently (in 1999) was released as a free software.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is *much* easier to program in a high-level language; by "easier" I mean that the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on 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 only used for a few special applications.

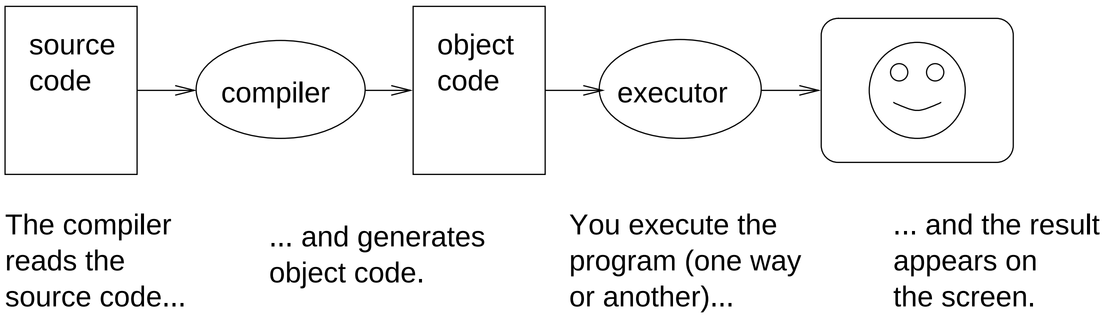
There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.


source code
interpreter
The interpreter reads the source code...
... and the result appears on the screen.

A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named `program.c`, where "program" is an arbitrary name you make up, and the suffix `.c` is a convention that indicates that the file contains C source code.

Then, depending on what your programming environment is like, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named `program.o` to contain the object code, or `program` to contain the executable.


source code
compiler
object code
executor
The compiler reads the source code...
... and generates object code.
You execute the program (one way or another)...
... and the result appears on the screen.

The advantage of compilation is that some computations can be done before running the program. Typically compiled programs run 10–100 times faster then interpreted ones. On the other hand compilation takes time so if we want to run program just once it may be faster to use interpreter than wait for compilation.

The Poplog system is an incremental compiler. For user it feels like an interpreter, but the commands given by the user are first compiled to machine code and the resulting machine code is directly executed by the processor. Since Poplog compiler is very fast (and works on small chunks of the program), from the point of view of the user commands are executed just after they are typed in.

Poplog also includes a more conventional batch compiler. This compiler works slower but can do slightly better job, so batch compiled programs gain some extra speed.

All this may seem complicated, the good news is that most of the time you will be using incremental compiler which hides most of the complexity under the hood. We can use incremental compiler in two ways. You can start the compiler from the command line and give it the name of the file containing your program. The compiler will then read your program, compile and then execute. You can also use compiler in the interactive mode, where you give commands and the compiler compiles and immediately executes them:

```
$ poplog pop11
Sussex Poplog Version 15.53

Sussex Poplog (Version 15.53 Wed Dec  8 23:03:00 CET 2004)
Copyright (c) 1982-1999 University of Sussex. All rights reserved.

Setpop
: 2 + 2 =>
** 4
:
```

Interactive mode is good for experimentation — normally you will keep programs in files, and submit small pieces to the compiler. It is workable to have compile and editor in separate windows and just cut part of your source file and paste it into the compiler, but there are also development environments (for example the ved editor coming with Poplog system) that largely automate this process — a single command allows you to compile and run part or whole program.

1.2 What is a program?

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

The instructions (or commands, or statements) look different in different programming languages, but there are a few basic functions that appear in just about every language:

input: Get data from the keyboard, or 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.

testing: 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 functions that look more or less like these. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple functions.

1.3 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.3.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, the error messages you get from the compiler are often not very helpful. 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, though, you will make fewer errors and find them faster.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program. In Poplog, run-time errors occur when you attempt to do some operation which makes no sense, like adding a number to a string:

```
: 1 + 'ala';

;;; MISHAP - NUMBER(S) NEEDED
;;; INVOLVING:  1 'ala'
;;; DOING    :  + pop_setpop_compiler

Setpop
:
```

In Poplog run-time errors are called mishaps (hence the message). In other languages they are usually called exceptions.

You are likely to see run-time errors even in very simple programs. The reason is that Poplog allows you to independently change parts of the program (even at runtime). This gives you much freedom during development. But this also means that Poplog can not check in advance if you program is consistent — inconsistent program may become consistent later.

You may be worried now, but the good news is that in simple programs it is easy to find the cause of a run-time error. You should then correct mistakes in the program and re-run it.

1.3.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will compile and 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.

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 logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire in this class 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 lead to the results you see.

Debugging is also like an experimental science. Once you have an idea 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." (from A. Conan Doyle's *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 always start with a working 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" (from *The Linux Users' Guide* Beta Version 1).

In later chapters I will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, like 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.**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3+3 = 6$ is a syntactically correct mathematical statement, but $3 = +6\$$ is not. Also, H_2O is a syntactically correct chemical name, but ${}_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with `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=+6$` is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

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 unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, 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 many 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 other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) 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, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally the first program people write in a new language is called "Hello, World." because all it does is print the words "Hello, World." In Pop11, this program looks like this:

```
printf('Hello World!\n');
```

Small explanation is in place: incremental compiler reads the program as a command and executes it. Semicolon indicates that the command is complete. Apostrophes indicate that the text between them is a string (which is passed as an `argument` to the `printf` procedure.

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, Pop11 does extremally well.

1.6 Glossary

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

high-level language: A programming language like Pop11 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."

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

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

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

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

compile: To translate a program in a high-level language 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 translating the program.

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

byte code: A special kind of object code used for Java programs. Byte code is similar to a low-level language, but it is portable, like a high-level language.

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

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

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

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

exception: An error in a program that makes it fail at run-time. Also called a run-time error.

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

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

Chapter 2

Variables, values and types

2.1 More on printing

2.1.1 Using `printf` procedure

You can put as many statements as you want in a program. For example, to print more than one line:

```
;;; generate some output
printf ('Hello, world.\n');     ;;; print one line
printf ('How are you?\n');      ;;; print another
```

Notice that the first line begins with `;;;`. This indicates that this line contains a **comment**, which is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a `;;;`, it ignores everything from there until the end of the line. Also, as you can see, it is legal to put comments at the end of a line, as well as on a line by themselves.

The phrases that are surrounded by apostrophes are called **strings**, because they are made up of a sequence (string) of letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

You may notice that strings that we passed to `printf` contain the funny character \ (called "backslash"). The combination of backslash and the letter `n` denotes **newline** — a special character that causes the cursor to move to the next line of the display.

One can put multiple newlines into a string, so that it prints as multiple lines:

```
;;; generate some output
printf ('Hello, world.\nHow are you?\n');  ;;; print two lines
```

Often it is useful to display the output from multiple print statements all on one line. This is easy: just add the newline only at the end of the last string:

```
;;; generate some simple output

printf ('Goodbye, ');
printf ('cruel world!\n');
```

In this case the output appears on a single line as `Goodbye, cruel world!`. Notice that there is a space between the word "Goodbye" and the second apostrophe. This space appears in the output, so it affects the behavior of the program.

Spaces that appear outside of quotation marks generally do not affect the behavior of the program. For example, I could have written:

```
  printf('Goodbye, ');
     printf('cruel world!\n');
```

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program's behavior either, so I could have written:

```
printf('Goodbye, '); printf('cruel world!\n');
```

That would work, too, although you have probably noticed that the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate syntax errors.

`printf` is short for "print formatted," because it not only prints strings, it may also convert (format) other values to strings. For example, I could write:

```
printf('Goodbye, cruel world!', '%p\n');
```

The last argument above is called "format string". It contains the sequence `%p` which in the output is replaced by the previous argument.

Printf is rather general function, here I will only show one more example:

```
printf(25, 5, '%p squared is %p\n');
```

which prints `5 squared is 25`.

2.1.2 Print arrow

Printf allow full control of the output, but requires some work to use. Since printing is frequently used for debugging, Pop11 contains a simpler construct: "print arrow". Writing:

```
'Goodbye, cruel world!' =>
```

prints `** Goodbye, cruel world!` in a separate line. Also note the two stars at the beginning of the line.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a named location that stores a **value**. Values are things that can be printed and stored and (as we'll see later) operated on. The strings we have been printing (`'Hello, World.'`, `'Goodbye, '`, etc.) are values.

In order to store a value, you have to create a variable.

```
lvars fred;
```

This statement is a **declaration**, because it declares that the word `fred` names a variable.

In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
lvars first_name;
lvars last_ame;
lvars hour, minute;
```

you could probably make a good guess at what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables: `hour` and `second`.

2.3 Assignment

Now that we have created some variables, we would like to store values in them. We do that with an **assignment statement**.

```
    'Hello.' -> fred;      ;;; give fred the value 'Hello.'
    11 -> hour;            ;;; assign the value 11 to hour
    59 -> minute;          ;;; set minute to 59
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

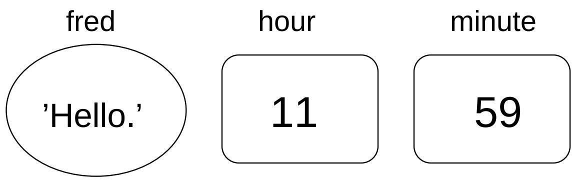
- When you declare a variable, you create a named storage location.
- When you make an assignment to a variable, you give it a value.

Unlike many other languages Pop11 allows you to store any value in a variable:

```
'123'  -> fred;   ;;; store a string
"fred" -> fred;   ;;; store a word
123    -> fred;   ;;; store an integer
[123]  -> fred;   ;;; store a list
{123}  -> fred;   ;;; store a vector
```

It may be confusing that some values look very similar, but are different. For example the integer `123` is different than the string `'123'` which in turn is different than the word `"'123'"`.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This figure shows the effect of the three assignment statements:



2.4 Printing variables

You can print the value of a variable using the same commands we used to print strings.

```
lvars first_line;
'Hello, again!' -> first_line;
first_line => ;;; print using print arrow
printf(first_line, '%p\n'); ;;; print again using printf
```

This program creates a variable named `first_line`, assigns it the value `'Hello, again!'` and then prints that value twice. When we talk about "printing a variable," we mean printing the *value* of the variable. To print the *name* of a variable, you have to put it in apostrophes. For example: `'first_line' =>`

If you want to get a little tricky, you could write

```
lvars first_line;
'Hello, again!' -> first_line;
printf ('The value of first_line is ');
printf (first_line, '%p\n');
```

The output of this program is

```
The value of first_line is Hello, again!
```

I am pleased to report that you can use the same the syntax for printing a value regardless of the value's type.

```
lvars hour, minute;
11 -> hour;
59 -> minute;
printf ('The current time is ', '%p');
printf (hour, '%p');
printf (':', '%p');
printf (minute, '%p');
printf ('.', '%p\n');
```

The output of this program is `The current time is 11:59.`

WARNING: It is common practice to use several `printf` commands, in order to put multiple values on the same line. But you have to be careful to remember the to add newline at the end.

2.5 Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are used by the Pop11 compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include `lvars`, `define`, `if`, `while`, and many more. Unlike most other languages Pop11 allows you add your own keywords or change some predefined keyword to an ordinary word.

Below we list some commonly used Pop11 keywords

and	by	constant	define	dlocal
do	else	elseif	enddefine	endfor
endif	endprocedure	endrepeat	endunless	enduntil
endwhile	for	from	goto	if
in	lconstant	lvars	nextif	nextloop
nextunless	or	procedure	quitif	quitloop
quitunless	repeat	return	step	then
to	unless	until	uses	vars

The complete list is contained in **ref** file `syntax`.

2.6 Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Many of the operators in Pop11 do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is `+`.

The following are all legal Pop11 expressions whose meaning is more or less obvious:

```
1+1        hour-1       hour*60 + minute     minute/60
```

Expressions can contain both variable names and numbers. In each case the name of the variable is replaced with its value before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
lvars hour, minute;
11 -> hour;
59 -> minute;
printf('Number of minutes since midnight: ');
printf(hour*60 + minute, '%p\n');
printf('Fraction of the hour that has passed: ');
printf(minute/60, '%p\n');
```

would generate the following output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 59_/60
```

The first line is what we expected, but the second line may be unexpected. The value of the variable `minute` is 59, and 59 divided by 60 is not an integer, so the result is a fraction 59/60. This is perfectly accurate, but different than typical answer form computer (or a pocket calculator).

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
printf('Percentage of the hour that has passed: ');
printf((minute*100) div 60, '%s\n');
```

The result is:

```
Percentage of the hour that has passed: 98
```

I have used `div` operator to round down the result, so it is only approximately correct. In order to get more accurate decimal fraction as an answer, we could use a different type of values, called floating-point, that is capable of storing fractional values. We'll get to that in the next chapter.

2.7 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but just to get you started:

- Multiplication take precedence (happen before) addition and subtraction. So `2*3-1` yields 5, not 4.
- Integer division `div` has higher precedence than multiplication. So `2*2 div 3` yields 0, not 1.
- If the operators have the same precedence they are evaluated from left to right. So in the expression `1-2+3`, the subtraction happens first, yielding `-1`, which in turn yields `2`. If the operations had gone from right to left, the result would be `1-5` which is `-4`, which is wrong.

- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so `2 * (3-1)` is 4. You can also use parentheses to make an expression easier to read, as in `minute * (100 div 60)`, even though it doesn't change the result.

2.8 Operators for strings

In general you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (if we know that fred contains a string)

```
fred - 1          'Hello'/123      fred * 'Hello'
```

Interestingly, there is an operator which work with strings: for strings, the `><` operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. So `'Hello, ' >< 'world.'` yields the string `'Hello, world.'` and `fred >< 'ism'` adds the suffix *ism* to the end of whatever `fred` is, which is often handy for naming new forms of bigotry.

2.9 Composition

So far we have looked at the elements of a programming language—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, we know how to multiply numbers and we know how to print; it turns out we can do both at the same time:

```
    printf (17 * 3, '%p\n');
```

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the printing, but the point is that any expression, involving numbers, strings, and variables, can be used inside a print statement. We've already seen one example:

```
printf(hour*60 + minute, '%p\n');
```

But you can also put arbitrary expressions on the left-hand side of an assignment statement:

```
lvars percentage;
(minute * 100) div 60 -> percentage;
```

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

WARNING: There are limits on where you can use certain expressions; most notably, the right-hand side of an assignment statement must have an "updater" which typically means that it has to be a *variable* name, not an expression. That's because the right-hand side indicates what to do with the result – typically the result is stored in a variable. So the following is illegal: `hour -> minute+1;`.

2.10 Glossary

variable: A named storage location for values. Normal Pop11 variables can store values of arbitrary type.

value: A number or string (or other thing to be named later) that can be stored in a variable. Every value belongs to some type.

type: A set of values. The type of a value determines which operation can be applied to the value. So far, the types we have seen are integers, rationals, strings and words.

keyword: A word that is used by the compiler to parse programs.

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

declaration: A statement that creates a new variable.

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

expression: A combination of variables, operators and values that represents a single result value. Expressions also have types, as determined by their operators and operands.

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.

precedence: The order in which operations are evaluated.

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

composition: The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

Chapter 3

Functions

3.1 Floating-point

In the last chapter we have seen that one can use fractions to deal with numbers that were not integers. While fractions give perfectly accurate results for arithmetic operations, and should be used when accuracy is a requirement, they are more complicated (both for humans and computer) than integers. Also, fractions can represent only rational numbers and many important mathematical functions (like square root) may give irrational result even for rational arguments. If we want numerical result from such a function the best we can do is to approximate the correct result. In other words the answer is strictly speaking wrong, but if the error is small enough this wrong answer works as well as the correct one.

In real life many quantities are known only approximately, and effort needed to get perfect accuracy during calculations is wasted. Scientists long ago invented a useful shortcut: decimal fractions (approximations) using exponential notation. It allows calculation using only few **significant digits** but usually gives reasonable result.

For approximate calculation on fractions computers use floating-point numbers. In Pop11, the floating-point type is called `decimal`, to stress connection with decimal fractions and exponential notation. However, on modern machines this name is a misnomer, because internally calculation are done in **binary**.

You can create floating-point variables and assign values to them using the same syntax we used for other values. For example:

```
lvars my_pi;
3.14159 -> my_pi;
```

It is also legal to declare a variable and assign a value to it at the same time:

```
lvars x = 1;
lvars empty = '';
lvars my_pi = 3.14159;
```

In fact, this syntax is quite common. A combined declaration and assignment is sometimes called an **initialization**. One can also combine multiple declarations and initializations in a single statement:

```
lvars x = 1, empty = '', my_pi = 3.14159;
```

Although floating-point numbers are useful, they are often a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value 1, is that an integer, a floating-point number, or both?

Strictly speaking, Pop11 distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number.

All the operations we have seen so far—addition, subtraction, multiplication, and division—also work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

3.2 Converting from decimals to integers

Pop11 converts an integer to decimal automatically if necessary, because the result is as accurate as possible. On the other hand, a decimal holds only an approximate value — even if this value looks like an integer we can not be sure that exact calculation would also produce an integer, so Pop11 does not automatically convert decimals to integers. Normally, converting floating point value to an integer requires rounding off.

The simplest way to convert a floating-point value to an integer is to use a `round` function:

```
lvars x = round(3.1415);
```

The `round` function has the effect of converting its argument into the closest integer, so `x` gets the value 3. If the closest integer is bigger than floating-point the value the rounded up, so for example `round(1.51)` gives 2.

If you want to throw out fractional part you can use `intof` function:

```
intof(3.99) =>
intof(-3.99) =>
```

The result is `3` and `-3`. Note, that for positive numbers `intof` rounds down while for negative numbers it rounds up.

3.3 Math functions

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (assuming that x is 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

Pop11 provides a set of built-in functions that includes most of the mathematical operations you can think of.

The math functions are invoked using a syntax that is similar to the `printf` commands we have already seen:

```
lvars root = sqrt (17.0);
lvars angle = 1.5;
lvars height = sin (angle);
```

The first example sets `root` to the square root of 17. The second example finds the sine of `1.5`, which is the value of the variable `angle`. By default Pop11 assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *degrees*. To work with radians you need to set magic variable `popradinas` to `true`:

```
true -> popradians;
```

To convert from degrees to radians, you can divide by 360 and multiply by 2π. Conveniently, Pop11 provides π as a built-in value:

```
lvars degrees = 90;
lvars angle = degrees * 2 * pi / 360.0;
```

We have already met another useful function, namely `round`, which rounds a floating-point value off to the nearest integer and returns an integer.

```
lvars x = round (pi * 20.0);
```

In this case the multiplication happens first, before the function is invoked. The result is 63 (rounded up from 62.8318).

3.4 Composition

Just as with mathematical functions, Pop11 functions can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
lvars x = cos (angle + pi/2);
```

This statement takes the value `pi`, divides it by two and adds the result to the value of the variable `angle`. The sum is then passed as an argument to the `cos` function.

You can also take the result of one function and pass it as an argument to another:

```
lvars x = exp (log (10.0));
```

In Pop11, the `log` function uses base e, so this statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

3.5 Adding new functions

So far we have only been using the functions that are built into Pop11, but it is also possible to add new functions:

```
define NAME ( LIST OF PARAMETERS );
    STATEMENTS
enddefine;
```

You can make up any name you want for your function, except that you can't use Pop11 keywords. The list of parameters specifies what information, if any, you have to provide in order to use (or **invoke**) the new function.

The first couple of functions we are going to write have no parameters, so the syntax looks like this:

```
define new_line();
    printf('\n');
enddefine;
```

This function is named `new_line`, and the empty parentheses indicate that it takes no parameters. It contains only a single statement, which prints a newline. Since newline skips to the next line after it prints, so this statement has the effect of skipping to the next line.

We can invoke this new function using syntax that is similar to the way we invoke the built-in Pop11 commands:

```
printf ('First line.\n');
new_line ();
printf ('Second line.\n');
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could invoke the same function repeatedly:

```
printf ('First line.\n');
new_line ();
new_line ();
new_line ();
printf ('Second line.\n');
```

Or we could write a new function, named `three_line`, that prints three new lines:

```
define three_line ();
    new_line (); new_line (); new_line ();
enddefine;

printf ('First line.\n');
three_line ();
printf ('Second line.\n');
```

You should notice a few things about this program:

- You can invoke the same procedure repeatedly. In fact, it is quite common and useful to do so.
- You can have one function invoke another function. In this case, main program invokes `three_line` and `three_line` invokes `new_line`. Again, this is common and useful.
- In `three_line` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new function gives you an opportunity to give a name to a group of statements. Functions can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code. Which is clearer, `new_line` or `printf ('\n')`?
2. Creating a new function can make a program smaller by eliminating repetitive code. For example, how would you print nine consecutive new lines? You could just invoke `three_line` three times.

3.6 Programs with multiple functions

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

```
define new_line();
    printf('\n');
enddefine;

define three_line();
    new_line (); new_line (); new_line ();
enddefine;

printf ('First line.\n');
three_line ();
printf ('Second line.\n');
```

When you look at a program that contains several functions, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution begins at the first statement at the top level (outside function), regardless of where it is in the program (in this case I deliberately put it at the bottom). Statements are executed one at a time, in order, until you reach a function invocation. Function invocations are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the invoked function, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one function can invoke another. Thus, while we are in the middle of the main program, we might have to go off and execute the statements in `three_line`. But while we are executing `three_line`, we get interrupted three times to go off and execute `new_line`.

For its part, `new_line` invokes the built-in function `printf`, which causes yet another detour. Fortunately, Pop11 is quite adept at keeping track of where it is, so when `printf` completes, it picks up where it left off in `new_line`, and then gets back to `three_line`, and then finally gets back to the main program so the program can terminate.

Actually, when given statements terminates it passes control back to interactive compiler which waits for next statement. If statements are read from the file at the end of the file the compiler cleans up and *then* the program terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

3.7 Parameters and arguments

Some of the built-in functions we have used have **parameters**, which are values that you provide to let the function do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a floating-point value as a parameter. To print a string, you have to provide the string, which is why `printf` takes a string as a parameter.

Some functions take more than one parameter, like `arctan2`.

New function also may have parameters:

```
define print_twice(phil);
    printf(phil, '%p\n');
    printf(phil, '%p\n');
enddefine;
```

This function takes a single parameter, named `phil`. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice. I chose the name `phil` to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than `phil`.

In order to invoke this function, we have to provide some value. For example, we might invoke it like this:

```
print_twice ('Don\'t make me say this twice!');
```

The string you provide is called an **argument**, and we say that the argument is **passed** to the function. In this case we are creating a string value that contains the text "Don't make me say this twice!" and passing that string as an argument to `print_twice` where, contrary to its wishes, it will get printed twice.

We can also use value of different type as an argument, for example we can pass an integer:

```
print_twice(123);
```

Alternatively, if we had a variable, we could use it as an argument instead:

```
lvars argument = 'Never say never.';
print_twice (argument);
```

Notice something very important here: the name of the variable we pass as an argument (`argument`) has nothing to do with the name of the parameter (`phil`). Let me say that again:

> **The name of the variable we pass as an argument has nothing to do with the name of the parameter.**

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the string `'Never say never.'`).

The value you provide as an argument may be of any type, however the called function must be able to handle it. Note that the compiler will accept argument of any type. However, he called function must be able to handle it — if the function receives argument if type which it can not handle you will get runtime error.

One last thing you should realize is that normally parameters exist only inside their own functions. In the main program, there is no such thing as `phil`. If you try to use it, the compiler will complain and create *a new different variable.*

3.8 Functions with multiple parameters

The syntax for declaring and invoking function with multiple parameters is similar to the one parameter case:

```
define print_time (hour, minute);
    printf (hour, '%p');
    printf (':');
    printf (minute, '%p\n');
enddefine;
```

The syntax to invoke our new function is `print_time (hour, minute)`.

3.9 Functions with results and the stack

You might have noticed by now that some of the functions we are using, like the mathematical functions, yield results. Other functions, like `prinf` and `new_line`, perform some action but they don't return a value. That raises some questions:

- What happens if you invoke a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a `prinf` function as part of an expression, like `printf ('boo!') + 7`?
- Can we write functions that yield results, or are we stuck with things like `new_line` and `print_twice`?

The answer to the third question is "yes, you can write functions that return values," and we'll do it in a couple of chapters. The answer to the other two questions is unexpected and slightly complicated.

Pop11 uses so called "stack" (or more precisely "user stack") to evaluate expression.

When you write a simple assignment like:

```
11 -> hours;
```

what happens is: the integer 11 is first put on the stack and then the value form the top of the stack is assigned to the variable `hours`. To make it more explicit you can write:

```
11;
-> hours;
```

Now, the first statement puts value on the stack and the second statement reads value form the top of the stack and assign to the variable `hours`.

3.10 Glossary

floating-point: A type of value that can contain fractions or integers. In Pop11 this type is called `decimal` (or `ddecimal`).

function: A named sequence of statements that performs something useful. Functions may or may not take parameters, and may or may not produce a result. Functions are frequently called procedures or subroutines.

parameter: A piece of information you provide in order to invoke a function. Parameters are like variables in the sense that they contain values.

argument: A value that you provide when you invoke a function.

invoke: Cause a function to be executed.

Chapter 4

Conditionals and recursion

4.1 The modulus and integer division

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In Pop11, the modulus operator is the keyword `rem`. The syntax is exactly the same as for other operators:

```
    7 div 3 -> quotient;
    7 rem 3 -> remainder;
```

The first operator, integer division `div`, yields 2. The second operator yields 1. Thus, 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 rem y` is zero, then `x` is divisible by `y`.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, `x rem 10` yields the rightmost digit of `x` (in base 10). Similarly `x rem 100` yields the last two digits.

4.2 Conditional execution

In order to write useful programs, we almost always need the ability to check certain 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
    printf('x is positive\n');
endif;
```

The expression between `if` and `then` is called the condition. If it is true, then the statements between `then` and `endif` get executed. If the condition is not true, nothing happens.

The condition can contain any of the comparison operators, sometimes called **relational operators**:

```
    x = y               ;;; x is equal to y
    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 syntax Pop11 uses is a little different from mathematical symbols like $\neq$ and $\leq$. Some other languages use a single `=` to denote assignment, which is a frequent source of errors. Also C-like languages use a double `==` for equality. In Pop11 `==` denotes identity (identical values are equal, but mathematically equal values do not need to be identical). Also, `=<` is the same as `<=`, but `=>` is a print arrow.

You can compare any two values for equality (or lack of equality), but the other four comparisons are valid only for numbers.

4.3 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if x rem 2 = 0 then
    printf ('x is even\n');
else
    printf ('x is odd\n');
endif;
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code prints a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a procedure, as follows:

```
define print_parity(x);
    if x rem 2 = 0 then
        printf ('x is even\n');
    else
        printf ('x is odd\n');
    endif;
enddefine;
```

Now you have a procedure named `print_parity` that will print an appropriate message for any integer you care to provide. You would invoke this procedure as follows:

```
print_parity (17);
```

4.4 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is to `chain` conditions using `elseif` keyword:

```
if x > 0 then
    printf ('x is positive\n');
elseif x < 0 then
    printf ('x is negative\n');
else
    printf ('x is zero\n');
endif;
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

4.5 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if x = 0 then
    printf ('x is zero\n');
else
    if x > 0 then
        printf ('x is positive\n');
    else
        printf ('x is negative\n');
    endif;
endif;
```

There is now an outer conditional that contains two branches. The first branch contains a simple `printf` statement, but the second branch contains another conditional statement, which has two branches of its own. Fortunately, those two branches are both `printf` statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

4.6 The return statement

The `return` statement allows you to terminate the execution of a procedure before you reach the end. One reason to use it is if you detect an error condition:

```
define print_logarithm (x);
    if x <= 0.0 then
        printf('Positive numbers only, please.\n');
        return;
    endif;
    lvars result = log(x);
    printf('The log of x is ' >< result >< '\n');
enddefine;
```

This defines a function named `print_logarithm` which has a parameter named `x`. The first thing it does is check whether `x` is less than or equal to zero, in which case it prints an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller and the remaining lines of the function are not executed.

I used a floating-point value on the right side of the condition because I expect that `x` has a floating-point value.

4.7 Type conversion

You might wonder how you can get away with an expression like `'The log of x is ' >< result`, since one of the operands is a string and the other is a `ddecimal`. Well, in this case Pop11 is being smart on our behalf, by automatically converting the `ddecimal` to a string before it does the string concatenation.

This kind of feature is an example of a common problem in designing a programming language, which is that there is a conflict between *formalism*, which is the requirement that formal languages should have simple rules with few exceptions, and *convenience*, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers (who are spared from rigorous but unwieldy formalism), but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

Nevertheless, it is handy to know that whenever you apply `><` to two expressions, then Pop11 will convert them to strings and then perform string concatenation.

4.8 Recursion

I mentioned in the last chapter that it is legal for one function to call another, and we have seen several examples of that. I neglected to mention that it is also legal for a function to invoke itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following function:

```
define countdown(n);
    if n = 0 then
        printf('Blastoff!\n');
    else
        printf(n, '%p\n');
        countdown (n - 1);
    endif;
enddefine;
```

The name of the function is `countdown` and it takes a single integer as a parameter. If the parameter is zero, it prints the word “Blastoff.” Otherwise, it prints the number and then invokes a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we invoke this function, in the main program, like this:

```
countdown (3);
```

The execution of `countdown` begins with `n=3`, and since `n` is not zero, it prints the value 3, and then invokes itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not zero, it prints the value 2, and then invokes itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not zero, it prints the value 1, and then invokes itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is zero, it prints 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` (what a trip). So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the functions `new_line` and `three_line`.

```
define new_line();
    printf('\n');
enddefine;

define three_line ();
    new_line (); new_line (); new_line ();
enddefine;
```

Although these work, they would not be much help if I wanted to print 2 newlines, or 106. A better alternative would be

```
define n_lines(n);
    if n > 0 then
        printf('\n');
        n_lines(n - 1);
    endif;
enddefine;
```

This program is very similar; as long as `n` is greater than zero, it prints one newline, and then invokes itself to print `n-1` additional newlines. Thus, the total number of newlines that get printed is `1 + (n-1)`, which usually comes out to roughly `n`.

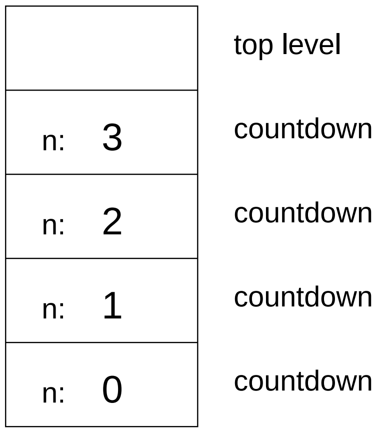
The process of a function invoking itself is called **recursion**, and such functions are said to be **recursive**.

4.9 Stack diagrams for recursive functions

In the previous chapter we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can make it easier to interpret a recursive function.

Remember that every time a function gets called it creates a new instance of the function that contains a new version of the function's local variables and parameters.

The following figure is a stack diagram for countdown, called with `n = 3`:



There is one instance of main program and four instances of `countdown`, each with a different value for the parameter `n`. The bottom of the stack, `countdown` with `n=0` is the base case. It does not make a recursive call, so there are no more instances of `countdown`.

The instance of main program is empty because the main program does not have any parameters or local variables. As an exercise, draw a stack diagram for `n_lines`, invoked with the parameter `n=4`.

4.10 Glossary

modulus: An operator that works on integers and yields the remainder when one number is divided by another. In Pop11 it is denoted by the `rem` word.

conditional: A block of statements that may or may not be executed depending on some condition.

chaining: A way of joining several conditional statements in sequence.

nesting: Putting a conditional statement inside one or both branches of another conditional statement.

recursion: The process of invoking the same function you are currently executing.

infinite recursion: A function that invokes itself recursively without every reaching the base case. The usual result is a runtime error.

Chapter 5

Fruitful functions

5.1 Return values

Some of the built-in functions we have used, like the mathematical functions, have produced results. That is, the effect of invoking the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
lvars e = exp (1.0);
lvars height = radius * sin (angle);
```

But so far all the function we have written have been **void** functions; that is, functions that return no value. When you invoke a void function, it is typically on a line by itself, with no assignment:

```
n_lines (3);
print_logarithm(17);
```

In this chapter, we are going to write functions that return things, which I will refer to as **fruitful** functions, for want of a better name. The first example is `area`, which takes a `decimal` as a parameter, and returns the area of a circle with the given radius:

```
define area(radius);
    lvars result = pi * radius * radius;
    return (result);
enddefine;
```

The last line is an `return` statement that includes a return value. This statement means, "return immediately from this function and use the following expression as a return value." Notice that the return value must be inside parenthesis. The expression you provide can be arbitrarily complicated, so we could have written this function more concisely:

```
define area(radius);
    return (pi * radius * radius);
enddefine;
```

On the other hand, **temporary** variables like `result` often make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
define absolute_value (x);
   if x < 0 then
       return(-x);
   else
       return(x);
   endif;
enddefine;
```

Since these returns statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one return statement in a function, you should keep in mind that as soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program returns correct value. For example:

```
define absolute_value (x);
   if x < 0 then
       return(-x);
   elseif x > 0 then
       return(x);
   endif;                 ;;; WRONG!!
enddefine;
```

This program is wrong because if `x` happens to be 0, then neither condition will be true and the function will end without hitting a return statement. Trying to use this nonexistent return value like:

```
absolute_value(0) + 1 =>
```

will produce error message like:

```
;;; MISHAP - ste: STACK EMPTY (missing argument? missing result?)
;;; DOING    :  + pop_setpop_compiler
```

Note that in Pop11 it is legal to return value for some arguments and no value for other arguments. In fact, it is possible to return multiple values, and the number of returned values may depend on the arguments. Similarly number of argument may be different for different invocations. This is a powerful feature, but is also error-prone — if a function is supposed to return a value but return statement is missing in correct place, then the error will be detected only at runtime (and sometimes such program may simply produce wrong values).

5.2 Program development

At this point you should be able to look at complete Pop11 functions and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest one technique that I call **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

The first step is to consider what a `distance` function should look like in Pop11. In other words, what are the inputs (parameters) and what is the output (return value).

In this case, the two points are the parameters, and it is natural to represent them using four `decimals`, although we will see later that one can also use a `Point` object. There is one return value, namely the distance.

Already we can write an outline of the function:

```
define distance (x1, y1, x2, y2);
    return (0.0);
enddefine;
```

The statement `return (0.0);` is a place-keeper that is necessary in order to compile the program. Obviously, at this stage the program doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we make it more complicated.

In order to test the new function, we have to invoke it with sample values. Somewhere in main program I would add:

```
lvars dist = distance (1.0, 2.0, 4.0, 6.0);
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a function, it is useful to know the right answer.

Once we have checked the syntax of the function definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. That way, at any point we know exactly where the error must be—in the last line we added.

The next step in the computation is to find the differences x_2-x_1 and y_2-y_1. I will store those values in temporary variables named `dx` and `dy`.

```
define distance (x1, y1, x2, y2);
    lvars dx = x2 - x1;
    lvars dy = y2 - y1;
    'dx is ' >< dx =>
    'dy is ' >< dy =>
    return (0.0);
enddefine;
```

I added print statements that will let me check the intermediate values before proceeding. As I mentioned, I already know that they should be 3.0 and 4.0.

When the function is finished I will remove the print statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square `dx` and `dy`. We could use the `**` operator, but it is simpler and faster to just multiply each term by itself.

```
define distance (x1, y1, x2, y2);
    lvars dx = x2 - x1;
    lvars dy = y2 - y1;
    lvars dsquared = dx*dx + dy*dy;
    'dsquared is ' >< dsquared =>
    return (0.0);
enddefine;
```

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use the `sqrt` function to compute and return the result.

```
define distance (x1, y1, x2, y2);
    lvars dx = x2 - x1;
    lvars dy = y2 - y1;
    lvars dsquared = dx*dx + dy*dy;
    lvars result = sqrt (dsquared);
    return (result);
enddefine;
```

Then in main program, we should print and check the value of the result.

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this incremental development process can save you a lot of debugging time.

The key aspects of the process are:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.
- Use temporary variables to hold intermediate values so you can print and check them.

- Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

5.3 Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a function, `distance` that does that.

```
lvars radius = distance (xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius, and return it.

```
lvars area1 = area (radius);
return (area1);
```

Wrapping that all up in a function, we get:

```
define area2 (xc, yc, xp, yp);
    lvars radius = distance (xc, yc, xp, yp);
    lvars area1 = area (radius);
    return (area1);
enddefine;
```

The name of this function is `area2`, because the name `area` is already taken. I will comment about this in the next section.

The temporary variables `radius` and `area1` are useful for development and debugging, but once the program is working we can make it more concise by composing the function invocations:

```
define area2 (xc, yc, xp, yp);
    return (area(distance (xc, yc, xp, yp)));
enddefine;
```

5.4 Overloading

In the previous section you might have noticed that `area2` and `area` perform similar functions—finding the area of a circle—but take different parameters. For `area`, we have to provide the radius; for `area2` we provide two points.

If two functions do the same thing, it is natural to give them the same name. In other words, it would make more sense if `area2` were called `area`.

Having more than one function with the same name, which is called **overloading**, is legal in Pop11, however there are certain restrictions, which are *not* satisfied in the current case. We will explain how overloading works in the chapter about object orientation. In Pop11 preferred style (and the only one possible without using object features) is to use longer names which uniquely identify functions. Choosing good names is an art, if names are hard to distinguish the programer may call different function than intended.

Actually, that reminds me of one of the cardinal rules of debugging: **make sure that the version of the program you are looking at is the version of the program that is running!** Some time you may find yourself making one change after another in your program, and seeing the same thing every time you run it. This is a warning sign that for one reason or another you are not running the version of the program you think you are. To check, stick in a `printf` statement (it doesn't matter what you print) and make sure the behavior of the program changes accordingly.

5.5 Boolean expressions

Most of the operations we have seen produce results that are the same type as their operands. For example, the `+` operator takes two integers and produces an integer, or two decimals and produces a decimal, etc.

The exceptions we have seen are the **relational operators**, which compare integers and floating point values and return either `true` or `false`. `true` and `false` are special values in Pop11, and together they make up a type called **boolean**. You might recall that when I defined a type, I said it was a set of values. In the case of integers, decimals and strings, those sets are pretty big. For `boolean`s, not so big.

Boolean expressions and variables work just like other types of expressions and variables:

```
lvars fred;
true -> fred;
lvars test_result = false;
```

The first example is a simple variable declaration; the second example is an assignment, and the third example is a combination of a declaration and as assignment, sometimes called an **initialization**. The values `true` and `false` are keywords in Pop11, so they may appear in a different color, depending on your development environment.

As I mentioned, the result of a conditional operator is a boolean, so you can store the result of a comparison in a variable:

```
lvars even_flag = (n rem 2 = 0);      ;;; true if n is even
lvars positive_flag = x > 0;     ;;; true if x is positive
```

and then use it as part of a conditional statement later:

```
if even_flag then
    printf('n was even when I checked it\n');
endif;
```

A variable used in this way is frequently called a **flag**, since it flags the presence or absence of some condition.

5.6 Logical operators

There are three **logical operators** in Pop11: AND, OR and NOT, which are denoted by the symbols `and`, `or` and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example `x > 0 and x < 10` is true only if `x` is greater than zero AND less than 10.

`even_flag or n rem 3 = 0` is true if *either* of the conditions is true, that is, if `even_flag` is true OR the number is divisible by 3.

Finally, the NOT operator has the effect of negating or inverting a boolean expression, so `not(even_flag)` is true if `even_flag` is false—if the number is odd. NOTE: the `not` is a function and *requires* parenthesis around its argument.

Logical operators often provide a way to simplify nested conditional statements. For example, how would you write the following code using a single conditional?

```
if x > 0 then
    if x < 10 then
        printf ('x is a positive single digit.\n');
    endif;
endif;
```

5.7 Boolean functions

Functions can return boolean values just like any other type, which is often convenient for hiding complicated tests inside functions. For example:

```
define is_single_digit (x);
    if x >= 0 and  x < 10 then
       return(true);
    else
       return(false);
    endif;
enddefine;
```

The name of this function is `is_single_digit`. It is common to give boolean functions names that sound like yes/no questions.

The code itself is straightforward, although it is a bit longer than it needs to be. Remember that the expression `x >= 0 and x < 10` has type boolean, so there is nothing wrong with returning it directly, and avoiding the `if` statement altogether:

```
define is_single_digit (x);
    return(x >= 0 and x < 10);
enddefine;
```

In the main program you can invoke this function in the usual ways:

```
lvars bib_flag = not(is_single_digit(17));
printf (is_single_digit (2), '%p\n');
```

The first line assigns the value `true` to `big_flag` only if 17 is *not* a single-digit number. The second line prints `<true>` because 2 is a single-digit number. Yes, `printf` can handle booleans, too.

The most common use of boolean functions is inside conditional statements

```
if is_single_digit (x) then
    printf ('x is little\n');
else
    printf ('x is big\n');
endif;
```

5.8 More recursion

Now that we have functions that return values, you might be interested to know that we have a **complete** programming language, by which I mean that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features we have used so far (actually, we would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that that claim is true is a non-trivial exercise first accomplished by Alan Turing, one of the first computer scientists (well, some would argue that he was a mathematician, but a lot of the early computer scientists started as mathematicians). Accordingly, it is known as the Turing thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

To give you an idea of what you can do with the tools we have learned so far, let's look at some functions for evaluating recursively-defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is typically not very useful:

frabjuous: an adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function **factorial**, you might get something like:

$$
\begin{aligned}
0! &= 1 \\
n! &= n \cdot (n-1)!
\end{aligned}
$$

(Factorial is usually denoted with the symbol !, which is not to be confused with the Pop11 operator `!` which has quite different meaning.) This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$. So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, we get 3! equal to 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Pop11 program to evaluate it. The first step is to decide what the parameters are for this function — factorial needs one parameter:

```
define factorial(n);
enddefine;
```

If the argument happens to be zero, all we have to do is return 1:

```
define factorial(n);
    if n = 0 then
        return(1);
    endif;
enddefine;
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$, and then multiply it by n.

```
define factorial(n);
    if n = 0 then
        return(1);
    else
        lvars recurse = factorial (n - 1);
        lvars result = n * recurse;
        return(result);
    endif;
enddefine;
```

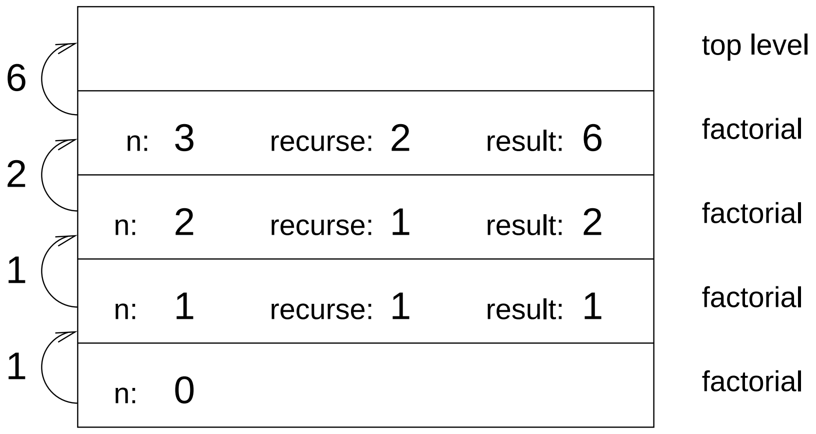
If we look at the flow of execution for this program, it is similar to `nLines` from the previous chapter. If we invoke `factorial` with the value 3:

Since 3 is not zero, we take the second branch and calculate the factorial of $n-1$...

> Since 2 is not zero, we take the second branch and calculate the factorial of $n - 1$...
>
> > Since 1 is not zero, we take the second branch and calculate the factorial of $n - 1$...
> >
> > > Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive calls.
> >
> > The return value (1) gets multiplied by `n`, which is 1, and the result is returned.
>
> The return value (1) gets multiplied by `n`, which is 2, and the result is returned.

The return value (2) gets multiplied by `n`, which is 3, and the result, 6, is returned to main program, or whoever invoked `factorial (3)`.

Here is what the stack diagram looks like for this sequence of function calls:



top level
6
n: 3 recurse: 2 result: 6
factorial
2
n: 2 recurse: 1 result: 2
factorial
1
n: 1 recurse: 1 result: 1
factorial
1
n: 0
factorial

The return values are shown being passed back up the stack.

Notice that in the last instance of `factorial`, the local variables `recurse` and `result` do not exist because when `n=0` the branch that creates them does not execute.

5.9 Leap of faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, it can quickly become labarynthine. An alternative is what I call the "leap of faith." When you come to a function invocation, instead of following the flow of execution, you *assume* that the function works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in functions. When you invoke `cos` or `printf`, you don't examine the implementations of those functions. You just assume that they work, because the people who wrote the built-in classes were good programmers.

Well, the same is true when you invoke one of your own functions. For example, in Section 5.7 we wrote a function called `is_single_digit` that determines whether a number is between 0 and 9. Once we have convinced ourselves that this function is correct—by testing and examination of the code—we can use the function without ever looking at the code again.

The same is true of recursive programs. When you get to the recursive invocation, instead of following the flow of execution, you should *assume* that the recursive invocation works (yields the correct result), and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it is a bit strange to assume that the function works correctly when you have not even finished writing it, but that's why it's called a leap of faith!

5.10 One more example

In the previous example I used temporary variables to spell out the steps, and to make the code easier to debug, but I could have saved a few lines:

```
define factorial(n);
    if n = 0 then
        return(1);
    else
        return(n * factorial (n-1));
    endif;
enddefine;
```

From now on I will tend to use the more concise version, but I recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up, if you are feeling inspired.

After `factorial`, the classic example of a recursively-defined mathematical function is `fibonacci`, which has the following definition:

$$
\begin{aligned}
fibonacci(0) &= 1\\
fibonacci(1) &= 1\\
fibonacci(n) &= fibonacci(n-1) + fibonacci(n-2);
\end{aligned}
$$

Translated into Pop11, this is

```
define fibonacci (n);
    if n = 0 or n = 1 then
```

```
        return(1);
    else
        return(fibonacci (n-1) + fibonacci (n-2));
    endif;
enddefine;
```

If you try to follow the flow of execution here, even for fairly small values of `n`, your head explodes. But according to the leap of faith, if we assume that the two recursive calls (yes, you can make two recursive calls) work correctly, then it is clear that we get the right result by adding them together.

5.11 Glossary

return value: The value provided as the result of a function invocation.

dead code: Part of a program that can never be executed, often because it appears after a `return` statement.

scaffolding: Code that is used during program development but is not part of the final version.

boolean: A type that contains only the two values `true` and `false`.

flag: A variable (usually `boolean`) that records a condition or status information.

conditional operator: An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator: An operator that combines boolean values and produces boolean values.

initialization: A statement that declares a new variable and assigns a value to it at the same time.

Chapter 6

Iteration

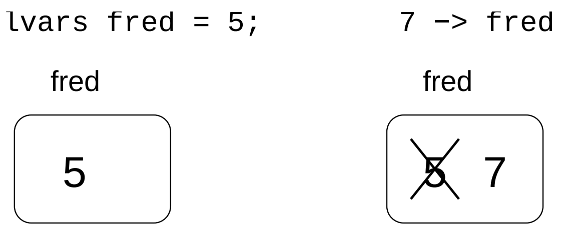
6.1 Multiple assignment

I haven't said much about it, but it is legal in Pop11 to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value.

```
lvars fred = 5;
printf(fred, '%p');
7 -> fred;
printf(fred, '%p\n');
```

The output of this program is `57`, because the first time we print `fred` his value is 5, and the second time his value is 7.

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:



When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Some other languages use the `=` symbol for assignment, which may cause confusion. Pop11 use `->` as an assignment symbol, which makes distinction very clear.

It is tempting to interpret a statement like `lvars a = b;` as a statement of equality. It kind of works if there is no more assignments to `a` and `b`. Namely, in mathematics, a statement of equality is true for all time. If $a = b$ now, then a will always equal b. In Pop11, even if variables are equal at one time they don't have to stay that way — an assignment later can make them unequal.

```
lvars a = 5;
lvars b = a;      ;;; a and b are now equal
3 -> a;           ;;; a and b are no longer equal
```

The third line changes the value of `a` but it does not change the value of `b`, and so they are no longer equal.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.

6.2 Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have already seen programs that use recursion to perform repetition, such as `n_lines` and `countdown`. This type of repetition is called **iteration**, and Pop11 provides several language features that make it easier to write iterative programs.

The two features we are going to look at are the `while` statement and the `for` statement.

6.3 The `while` statement

Using a `while` statement, we can rewrite `countdown`:

```
define countdown(n);
    while n > 0 do
        printf(n, '%p\n');
        n - 1 -> n;
    endwhile;
    printf ('Blastoff!');
enddefine;
```

You can almost read a `while` statement as if it were English. What this means is, "While `n` is greater than zero, continue printing the value of `n` and then reducing the value of `n` by 1. When you get to zero, print the word 'Blastoff!"'

More formally, the flow of execution for a `while` statement is as follows:

1. Evaluate the condition between `while` and `do`, yielding `true` or `false`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.

3. If the condition is true, execute each of the statements between the `do` and `endwhile`, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The statements inside the loop are sometimes called the **body** of the loop.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite** loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `countdown`, we can prove that the loop will terminate because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop (each **iteration**), so eventually we have to get to zero. In other cases it is not so easy to tell:

```
define sequence(n);
    while n /= 1 do
        printf(n, '%p\n');
        if n rem 2 = 0 then  ;;; n is even
            n div 2 -> n;
        else                 ;;; n is odd
            n*3 + 1 -> n;
        endif;
    endwhile;
enddefine;
```

The condition for this loop is `n /= 1`, so the loop will continue until `n` is 1, which will make the condition false.

At each iteration, the program prints the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program will terminate. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it!

6.4 Tables

One of the things loops are good for is generating and printing tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand.

To make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the result tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly), but shortsighted. Soon thereafter computers (and calculators) were so pervasive that the tables became obsolete.

Well, almost. It turns out that for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division.

Although a "log table" is not as useful as it once was, it still makes a good example of iteration. The following program prints a sequence of values in the left column and their logarithms in the right column:

```
lvars x = 1.0;
while x < 10.0 do
    printf(x, '%p   ');
    printf(log(x), '%p\n');
    x + 1.0 -> x;
endwhile;
```

The output of this program is

```
1.0   0.0
2.0   0.693147
3.0   1.09861
4.0   1.38629
5.0   1.60944
6.0   1.79176
7.0   1.94591
8.0   2.07944
9.0   2.19722
```

Looking at these values, can you tell what base the `log` function uses by default?

Since powers of two are so important in computer science, we often want to find logarithms with respect to base 2. To find that, we have to use the following formula:

$$\log_2 x = \frac{log_e x}{log_e 2} \tag{6.1}$$

Changing the `printf` statement containing log to

```
printf (log(x) / log(2.0), '%p\n');
```

yields

```
1.0   0.0
2.0   1.0
3.0   1.58496
4.0   2.0
5.0   2.32193
6.0   2.58496
7.0   2.80735
8.0   3.0
9.0   3.16992
```

We can see that 1, 2, 4 and 8 are powers of two, because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
lvars x = 1.0;
while x < 100.0 do
    printf(x, '%p   ');
    printf (log(x) / log(2.0), '%p\n');
    x * 2.0 -> x;
endwhile;
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a **geometric** sequence. The result is:

```
1.0   0.0
2.0   1.0
4.0   2.0
8.0   3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

Log tables may not be useful any more, but for computer scientists, knowing the powers of two is! Some time when you have an idle moment, you should memorize the powers of two up to 65536 (that's 2^{16}).

6.5 Two-dimensional tables

A two-dimensional table is a table where you choose a row and a column and read the value at the intersection. A multiplication table is a good example. Let's say you wanted to print a multiplication table for the values from 1 to 6.

A good way to start is to write a simple loop that prints the multiples of 2, all on one line.

```
lvars i = 1;
while i <= 6 do
    printf (2*i, '%p   ');
    i + 1 -> i;
endwhile;
printf('\n');
```

The first line initializes a variable named `i`, which is going to act as a counter, or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6, and then when `i` is 7, the loop terminates. Each time through the loop, we print the value `2*i` followed by three spaces. Since we print newline only in the final statement, all the output appears on a single line.

The output of this program is:

```
2   4   6   8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

6.6 Encapsulation and generalization

Encapsulation usually means taking a piece of code and wrapping it up in a function, allowing you to take advantage of all the things functions are good for. We have seen two examples of encapsulation, when we wrote `print_parity` in Section 4.3 and `is_single_digit` in Section 5.7.

Generalization means taking something specific, like printing multiples of 2, and making it more general, like printing the multiples of any integer.

Here's a function that encapsulates the loop from the previous section and generalizes it to print multiples of `n`.

```
define print_multiples(n);
    lvars i = 1;
    while i <= 6 do
        printf (n*i, '%p   ');
        i + 1 -> i;
    endwhile;
    printf('\n');
enddefine;
```

To encapsulate, all I had to do was add the first line, which declares the name and parameter, and the last line. To generalize, all I had to do was replace the value 2 with the parameter `n`.

If I invoke this function with the argument 2, I get the same output as before. With argument 3, the output is:

```
3    6    9    12    15    18
```

and with argument 4, the output is

```
4    8    12    16    20    24
```

By now you can probably guess how we are going to print a multiplication table: we'll invoke `print_multiples` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

```
lvars i = 1;
while i <= 6 do
    print_multiples (i);
    i + 1 -> i;
endwhile;
```

First of all, notice how similar this loop is to the one inside `print_multiples`. All I did was replace the print statement with a function invocation.

The output of this program is

```
1    2    3    4    5    6
2    4    6    8    10    12
3    6    9    12    15    18
4    8    12    16    20    24
5    10    15    20    25    30
6    12    18    24    30    36
```

which is a (slightly sloppy) multiplication table. If the sloppiness bothers you, Pop11 provides functions that give you more control over the format of the output, but I'm not going to get into that here.

6.7 Functions

In the last section I mentioned "all the things functions are good for." About this time, you might be wondering what exactly those things are. Here are some of the reasons functions are useful:

- By giving a name to a sequence of statements, you make your program easier to read and debug.
- Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.

- Functions facilitate both recursion and iteration.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

6.8 More encapsulation

To demonstrate encapsulation again, I'll take the code from the previous section and wrap it up in a function:

```
define print_mult_table();
    lvars i = 1;
    while i <= 6 do
        print_multiples (i);
        i + 1 -> i;
    endwhile
enddefine;
```

The process I am demonstrating is a common development plan. You develop code gradually by adding lines to main program or someplace else, and then when you get it working, you extract it and wrap it up in a function.

The reason this is useful is that you sometimes don't know when you start writing exactly how to divide the program into functions. This approach lets you design as you go along.

6.9 Local variables

About this time, you might be wondering how we can use the same variable `i` in both `print_multiples` and `print_mult_table`. Didn't I say that you can only declare a variable once? And doesn't it cause problems when one of the functions changes the value of the variable?

The answer to both questions is "no," because the `i` in `print_multiples` and the `i` in `print_mult_table` are *not the same variable.* They have the same name, but they do not refer to the same storage location, and changing the value of one of them has no effect on the other.

Variables that are declared inside a function definition are called **local variables** because they are local to their own functions. You cannot access a local variable from outside its "home" function, and you are free to have multiple variables with the same name, as long as they are not in the same function.

It is often a good idea to use different variable names in different function, to avoid confusion, but there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

6.10 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to print_mult_table:

```
define print_mult_table(high);
    lvars i = 1;
    while i <= high do
        print_multiples (i);
        i + 1 -> i;
    endwhile
enddefine;
```

I replaced the value 6 with the parameter high. If I invoke print_mult_table with the argument 7, I get

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
7   14   21   28   35   42
```

which is fine, except that I probably want the table to be square (same number of rows and columns), which means I have to add another parameter to print_multiples, to specify how many columns the table should have.

Just to be annoying, I will also call this parameter high, demonstrating that different functions can have parameters with the same name (just like local variables):

```
define print_multiples(n, high);
    lvars i = 1;
    while i <= high do
        printf (n*i, '%p   ');
        i + 1 -> i;
    endwhile;
    printf('\n');
enddefine;

define print_mult_table(high);
    lvars i = 1;
    while i <= high do
        print_multiples (i);
        i + 1 -> i;
    endwhile
enddefine;
```

Notice that when I added a new parameter, I had to change the first line of the function (containing parameter list), and I also had to change the place where the function is invoked in `printi_mult_table`. As expected, this program generates a square 7x7 table:

```
1    2    3    4    5    6    7
2    4    6    8    10    12    14
3    6    9    12    15    18    21
4    8    12    16    20    24    28
5    10    15    20    25    30    35
6    12    18    24    30    36    42
7    14    21    28    35    42    49
```

When you generalize a function appropriately, you often find that the resulting program has capabilities you did not intend. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `print_mult_table`. Change

```
      print_multiples (i, high);
```

to

```
      print_multiples (i, i);
```

and you get

```
1
2    4
3    6    9
4    8    12    16
5    10    15    20    25
6    12    18    24    30    36
7    14    21    28    35    42    49
```

I'll leave it up to you to figure out how it works.

6.11 Glossary

loop: A statement that executes repeatedly while or until some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

encapsulate: To divide a large complex program into components (like functions) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a function and that exists only within that function. Local variables cannot be accessed from outside their home function, and do not interfere with any other functions.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing. In Section 5.2 I demonstrated a technique I called incremental development. In later chapters I will suggest other styles of development.

Chapter 7

Strings and things

7.1 Characters and codes

The data contained in a string are the letters of the string. Individual letters are called characters. In Pop11 characters are *not* a separate type, each character is represented by a small integer, called "character code". The exact correspondence between characters and integer is called an "character encoding" and vary widely between computer systems. Fortunately, there is subset of characters which is encoded the same on all computers running Pop11 — called the ASCII charset. All characters that we will use are in this subset.

How Pop11 know that an integer is really not an integer, but represents a character? Well, it does not know, Pop11 just performs operation on integers and most of the time this gives intended effect. For example, to copy a character just copy corresponding integer. However, sometimes you need tell Pop11 that you want an integer to be handled as a character. For example, integer 97 represents lowercase 'a'. To print it and see the character we have to tell `printf` that we want a character. Compare:

```
printf(97, '%p\n');
printf(97, '%c\n');
```

The first statement prints `97` while the second prints `a`. I wrote previously that print replaces `%p` sequence in the second argument by its first argument. In fact, printf converts its first argument to a string and then replaces `%p` by the resulting string. For each type Pop11 has rules how to convert it to a string.

The `%c` sequence is similar to `%p`, but requires integer argument and simply treats it as a character code.

It would be awkward to write such numbers as 97 in programs — Pop11 has special notation for writing characters: just put a character inside grave quotes and Pop11 will know that you want this character code:

```
lvars fred = `c`;
```

```
printf(fred, '%p\n');
printf(fred, '%c\n');
```

Unlike string values (which appear in apostrophes), character values can contain only a single letter.

7.2 Extracting character from a string

Pop11 uses parenthesis `(` and `)` to extract characters from strings:

```
lvars fruit = 'banana';
lvars letter = fruit(1);
printf (letter, '%c\n');
```

The expression `fruit(1)` indicates that I want character number 1 (that is first character) from the string named `fruit`. Note: many languages start counting from 0, however counting starting from 1 is more natural for humans.

Note that `fruit(1)` looks like a function invocation — you may wonder why Pop11 uses the same notation here. In fact, `fruit(1)` *is* a function invocation: in Pop11 string may serve as a function and applying this function to an integer `n` gives character number `n` from the string.

7.3 Length

The second function we'll look at is `length`, which returns the number of characters in the string. For example:

```
lvars fruilt_length = length(fruit);
```

`length` takes the string as an argument and returns an integer, in this case 6.

To find the last letter of a string, you may do

```
lvars fruilt_length = length(fruit);
lvars last = fruit(length(fruit));
```

7.4 Traversal

A common thing to do with a string is 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**. A natural way to encode a traversal is with a `while` statement:

```
lvars index = 1;
while index <= length(fruit) do
   lvars letter = fruit(index);
   printf(letter, '%c\n');
   index + 1 -> index;
endwhile;
```

This loop traverses the string and prints each letter on a line by itself. Notice that the condition is `index <= length(fruit)`, which means that when `index` is bigger than the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `length(fruit)`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set (in this case the set of characters in the string). The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a function that takes a string as an argument and that prints the letters backwards, all on one line.

7.5 Run-time errors

Way back in Section 1.3.2 I talked about run-time errors, which are errors that don't appear until a program has started running.

7.6 Reading documentation

7.7 The `locchar` function

In some ways, `locchar` is the opposite of the `()` operator. `()` takes an index and returns the character at that index. `locchar` takes a character and finds the index where that character appears.

`locchar` fails if the character does not appear in the string, and returns the value `false`.

```
lvars fruit = 'banana';
lvars index = locchar(`a`, 1, fruit);
```

This finds the index of the letter `'a'` in the string. In this case, the letter appears three times, so it is not obvious what `locchar` should do. According to the documentation, it returns the index of the *first* appearance. You may ask why we gave three arguments to `locchar`: the second argument tells `locchar` where to start searching (in our case at index 1).

In order to find subsequent appearances of 'a' invoke

```
lvars index = locchar(`a`, 3, fruit);
```

it will start at the third letter (the first `n`) and find the second `a`, which is at index 4. If the letter happens to appear at the starting index, the starting index is the answer. Thus,

```
lvars index = locchar(`a`, 6, fruit);
```

returns 6.

What happens if the starting index is out of range: if the starting index is negative or zero you will get runtime error. If the starting index is bigger than the length of string `locchar` gives you `false` indicating that the character is not found.

7.8 Looping and counting

The following program counts the number of times the letter `'a'` appears in a string:

```
lvars fruit = 'banana';
lvars fruilt_length = length(fruit);
lvars count = 0;
lvars index = 1;
while index <= fruilt_length do
    if fruit(index) = `a` then
        count + 1 -> count;
    endif;
    index + 1 -> index;
endwhile;
printf (count, '%p\n');
```

This program demonstrates a common idiom, called a **counter**. The variable `count` is initialized to zero and then incremented each time we find an `` `a` `` (to **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to **excrement**, which is a noun). When we exit the loop, `count` contains the result: the total number of a's.

As an exercise, encapsulate this code in a function named `count_letters`, and generalize it so that it accepts the string and the letter as arguments.

As a second exercise, rewrite the method so that it uses `locchar` to locate the a's, rather than checking the characters one by one.

7.9 Creating strings from characters

Often we have a few charaters in hand and want to create from then a new string. Pop11 has a special for that purpose: `consstring`. For example

```
consstring(`O`, `K`, 2) =>
```

Prints OK. The consstring function is somewhat unusual, it takes variable number of arguments. It one argument for each character in the string and the last argument tells consstring how long the string is, hence, the number of other arguments. Since OK has length 2 we passed 2 as the last argument.

If the string is constant putting it in apostrophes is more convenient than using consstring, but consstring works also when charaters are stored in variables

```
lvars g = ‘G‘;
lvars o = ‘o‘;
consstring(g, o, o, ‘d‘, 4) =>
```

7.10 Strings are mutable

The () operator not only allows you to access individual characters of the string, it also allows you to change them. For example

```
lvars fruit = ’banana’;
‘f‘ -> fruit(1);
printf(fruit, ’%p\n’);
```

prints fanana.

7.11 Character arithmetic

Since in Pop11 character really *are* you can do arithmetic with characters! The expression ‘a‘ + 1 yields the same value as ‘b‘. Similarly, if you have a variable named letter that contains a character, then letter - ’a’ + 1 will tell you where in the alphabet it appears.

This sort of thing is useful for converting between the characters that contain numbers, like ‘0‘, ‘1‘ and ‘2‘, and the corresponding integers. To convert ’3’ to the corresponding integer value you can subtract ’0’:

```
lvars x = letter - ‘0‘;
```

Another use for character arithmetic is to loop through the letters of the alphabet in order. For example, in Robert McCloskey’s book *Make Way for Ducklings*, the names of the ducklings form an abecedarian series: Jack, Kack, Lack, Mack, Nack, Ouack, Pack and Quack. Here is a loop that prints these names in order:

```
lvars name = ’Jack’;
lvars letter = ‘J‘;
while letter <= ‘Q‘ do
    letter -> name(1);
    printf (name, ’%p\n’);
    letter + 1 -> letter;
endwhile;
```

Notice that in addition to the arithmetic operators, we can also use the conditional operators on characters. The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because I've misspelled "Ouack" and "Quack." As an exercise, modify the program to correct this error.

Above, we used modified the value of `name`. We could alternatively create new string in each iteration. To do this it is enough to replace the line

```
    letter -> name(1);
```

by

```
    consstring(letter, 1) >< 'ack' -> name;
```

7.12 Generalized booleans

Up to now we used the `if` statement many times. You may thin that you need boolean expression as a condition in the `if` statement. This is not true. Pop11 has so called "generalized booleans". It allows any value in condition. The boolean `false` means really false, while any other value is treated as true

```
if 0 then
    printf('0 is the same as true\n');
endif;
```

Prints `0 is the same as true`.

Generalized booleans frequently allow writing shorter and more efficient programs, but they may be confusing, especially for beginners. In this book I will try to avoid them, but many built-in Pop11 functions return generalized booleans, so you need to know about them. Let me also say that logical operators `not`, `and` and `or` handle generalized booleans.

7.13 Comparing strings

It is often necessary to compare strings to see if they are the same, or to see which comes first in alphabetical order. To check for equality we can just use `=` or `/=` operator. Later you will learn about `==` operator—typically you do *not* want to use this operator to compare strings. It would be nice if we could use other comparison operators, like `>`, but we can't.

In order check which string comes first in dictionary order use `alphabefore` function. For example:

```
lvars name1 = 'Alan Turing';
lvars name2 = 'Ada Lovelace';

if name1 = name2 then
    printf ('The names are the same.\n');
elseif alphabefore(name1, name2) then
    printf ('name1 comes before name2.\n');
else
    printf ('name2 comes before name1.\n');
endif;
```

The return value from `alphabefore` is strange: `true` if the first string is lexically before the second, `false` if the second string is lexically before the first and `1` if both strings contain the same characters. So the return value of `alphabefore` is a generalized boolean.

Using properties of `alphabefore` we can do comparison in slightly different way:

```
lvars name1 = 'Alan Turing';
lvars name2 = 'Ada Lovelace';
lvars cmp = alphabefore(name1, name2);

if cmp = 1 then
    printf ('The names are the same.\n');
elseif cmp then
    printf ('name1 comes before name2.\n');
else
    printf ('name2 comes before name1.\n');
endif;
```

7.14 Glossary

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

traverse: To iterate through all the elements of a set performing a similar operation on each.

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

increment: Increase the value of a variable by one.

decrement: Decrease the value of a variable by one.

exception: A run time error. Exceptions cause the execution of a program to terminate.

Chapter 8

Words and lists

8.1 Identity

I wrote that values are stored in variables. It is more precise to say that variables "reference" values. Namely, two different variables can reference ("contain") the same value. Look at the following example

```
lvars fred = 'day';
lvars frank = fred;
`s` -> frank(1);
printf(fred, '%p\n');
```

It prints `say`. One would expect that `fred` contains the string `day` and that this sting should appear in the output. What happened? The second line copy *reference* but the string itself is *not* copied, so both `fred` and `frank` reference *the same* string. Hence, modifying string referenced by `frank` has effect on `fred`.

How can you detect that two variables reference the same (identical) value and not merely an exact copy? Pop11 uses `==` operator to check if two values are identical. For example

```
lvars fred = 'day';
lvars frank = fred;
frank == fred =>
'day' -> frank;
frank == fred =>
frank = fred =>
```

prints `<true>`, `<false>` and `<true>`. After the second line `fred` and `frank` are identical. The line `'day' -> frank;` assings a new string to `frank`. This new string has the same charactes as the old one, so `frank` and `fred` are equal, but not identical.

Checking for identity is very cheap: this check essentially compares it two values are stored in the same memory location—this can be done in a single machine operation. On the other hand checking equality may be much more expensive: to check that two strings are equal one has to compare all characters are equal, which requires several machine operations. To get better execution speed some programmers prefer to use identity of values instead of equality. However, real life problem usually require equality. We could get the best of two words if could arrange things such that equal values are always identical (that is stored in the same memory location). In general, trying to make equal values identical has many problems. But we can get most of benefits if we replace strings by words.

8.2 Words

Words is special data type related to strings. However, unlike strings equal words are always identical. To guarantee that equal words are identical Pop11 stores all words in a special central place, called **dictionary**. When you try to create a new word Pop11 first checks if the word is present in the dictionary. If the word is present in the dictionary Pop11 uses the copy from dictionary. If the word is absent from the dictionary Pop11 add the new word to the dictionary. Words can be used similarly to strings.

To create a new word we put the letter in quotes:

```
lvars word1 = "I_am_a_word";
```

Note: if you want to have some funny characters inside a word you need to first put characters into apostrophes (like string) and then put quotes around it:

```
lvars word2 = "'I am a word'";
```

Print arrow and `printf` function works on words

```
word1 =>
printf(word1, '%p\n');
```

Note: you can not use one argument version of `printf` to print words, because the *last* argument to `printf` must be a string.

As I promised, equal words are identical

```
lvars word3 = "I_am_a_word";
word1 == word3 =>
```

Given a string we can create corresponding word using `consword`:

```
lvars word = consword('word from string');
```

The `consword` function can also create a word from characters

```
lvars word = consword(`O`, `K`, 2);
```

We can use `word_string` to convert a word to a string

```
lvars string = word_string(word);
```

8.3 Type predicates

As saw in the previous section words and strings when printed look exactly the same. So have can we check that given variable references a word (as opposed say to a string). Pop11 has a general solution to this problem. Each data type has a special function which returns `true` if a value is of that type and `false` otherwise. Such function is called "recognizer" (or "type predicate"). In Pop11 type predicates have very regular names: the name of type predicate is the name of the type with the letters `is` prepended. So, to check is a value is a word we use `isword` function

```
isword(word) =>
```

To check for strings we use `isstring` function

```
isstring(word) =>
isstring('a string') =>
```

8.4 Creating lists

We can create a list by writting some elements inside the square brackets `[` and `]`:

```
lvars numbers = [1 2 5];
numbers =>
```

which prints `[1 2 5]`.

Inside list brackets Pop11 uses special rules to read the program

```
lvars fruits = [apple banana pear];
fruits =>
```

gives a list of three words – it like all elements were in quotes.

We can put elements of any type in a list, even **nested** lists

```
lvars mixed = [1 apple 'banana' [4 5]];
mixed =>
```

gives a list having the integer `1` as the first element, the word `apple` as the second element, the string `banana` as the thired element and the list `[1 2]` as the fourth element.

8.5 Extracting element from a list

Pop11 uses `first` function to extact first element from a list

```
first(fruits) =>
```

To extract any element from lists, Pop11 uses parenthesis `(` and `)`

```
lvars fruit = fruits(2);
printf (fruit, '%p\n');
```

The expression `fruits(2)` indicates that I want element number 2 (that is second element) from the list named `fruits`.

NOTE: `fruits(n)` looks like a simple operation, but in fact it must traverse the list and skip first `n - 1` elements. For large `n` it is a substantial work.

8.6 Length

The `length` returns the number of characters in the list. For example:

```
lvars fruilts_length = length(fruits);
```

We alredy saw that `length` can compute length of a string, now we see that it computes length of a list. In fact, `length` is a general function which can be applied to many different types.

8.7 Linked lists

Pop11 lists are **linked lists**. The simplest possible list is an **empty list**, which contains no elements. Pop11 uses square brackets with on elements to denote empty list: `[]`. Other lists consist of series of **nodes**, each one references the next one (the last node references the empty list). Additionally, each node stores a value. The `first` function extracts values stored in the first node. The list referenced by the first node is called **tail** of a list. Pop11 uses the `back` function to get the tail

```
back([first second third]) =>
```

You can use `conspair` function to build list with given first element and tail. For example, if tail is the empty list we get one element list

```
conspair(2, []) =>
```

We can build longer lists composing `conspair`

```
conspair("a", conspair("longer", conspair("list", []))) =>
```

8.8 List traversal

A common thing to do with a list is start at the beginning, select each element in turn, do something to it, and continue until the end. A natural way to handle lists is via recursion: first we handle the first element (given by the `first`) function, then we apply the same operation to the tail (given by the `back` function). We stop when we arrive to the empty list.

However, it is easy to do the same using `while` statement:

```
while fruits /= [] do
   lvars element = front(fruits);
   printf(element, '%p\n');
   back(fruits) -> fruits;
endwhile;
```

This loop traverses the string and prints each element on a line by itself. Notice that the condition is `fruits /= []`, which means that when `fruits` is empty than, the condition is false and the body of the loop is not executed.

Traversing list is a common operation so Pop11 provides special syntax for this purpose, the `for` loop

```
lvars element;
for element in fruits do
    printf(element, '%p\n');
endfor;
```

8.9 List are mutable

We can modify list elements:

```
lvars animals = [rat dog cat];
animals =>
"horse" -> first(animals);
animals =>
"snake" -> animals(3);
animals =>
```

8.10 Glossary

Chapter 9

Compound values

9.1 Why to use objects?

A common motivation for creating a new object type is to take several related pieces of data and encapsulate them into an object that can be manipulated (passed as an argument, operated on) as a single unit.

Although `String`s and lists are compound values, their components have no name. Sometimes unnamed components are a feature, however typically named components make program easier to read and write.

Here are the most important ideas in this chapter:

- A class definition is like a template for objects: it determines what slots (instance variables) the objects have.

- Every object belongs to some object type; hence, it is an instance of some class.

- When you define a class like `Point` Pop11 automatically creates several functions for you: slot **accessors**, and **constructors**. Slot accessorss have the same name as slots. Name of constructors is the name of class with the word `cons` or `new` prepended – the `Point` class have `consPoint` and `newPoint` constructors.

- You create a new object of a class by calling its constructor.

- You read or modify slots using slot accessors.

In this chapter, we are going to create two new object types `Point` and `Rectangle`. Right from the start, I want to make it clear that these points and rectangles are not graphical objects that appear on the screen. They are variables that contain data, just like `int`s and `double`s. Like other variables, they are used internally to perform computations.

9.2 Libraries

Poplog system is divide into many parts called **libraries**. Pop11 support object is in a library called `objectclass`. Before we can use object we must first tell Pop11 that we need the library using `uses` statement:

```
uses objectclass;
```

9.3 `Point` objects

At the most basic level, a point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

To represent points in Pop11 we will create `Point` class.

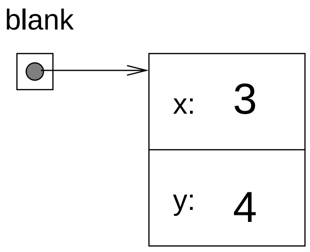
```
define :class Point;
   slot x = 0;
   slot y = 0;
enddefine;
```

To create a new point, you can use the `consPoint` function:

```
lvars blank = consPoint (3, 4);
```

This line is a conventional variable declaration: `blank` is initialized with the value produced by `consPoint`. It will probably not surprise you that the arguments to `consPoint` are the coordinates of the new point, $(3, 4)$. In general `consSomething` need an argument for each slot (instance variable) of `Something` (the arguments appear in the same order as slots in the class definition).

The result of the `consPoint` function is a **reference** to the new point. I'll explain references more later; for now the important thing is that the variable `blank` contains a reference to the newly-created object. There is a standard way to diagram this assignment, shown in the figure.



As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a reference, which is shown graphically with a dot and an arrow. The arrow points to the object we're referring to.

The big box shows the newly-created object with the two values in it. The names `x` and `y` are the names of the **instance variables**.

Taken together, all the variables, values, and objects in a program are called the **state**. Diagrams like this that show the state of the program are called **state diagrams**. As the program runs, the state changes, so you should think of a state diagram as a snapshot of a particular point in the execution.

9.4 Instance variables

The pieces of data that make up an object are sometimes called components, records, or fields. In Pop11 the they are officially called slots, but we will call them instance variables because each object, which is an **instance** of its type, has its own copy of the instance variables.

It's like the glove compartment of a car. Each car is an instance of the type "car," and each car has its own glove compartment. If you asked me to get something from the glove compartment of your car, you would have to tell me which car is yours.

Similarly, if you want to read a value from an instance variable, you have to specify the object you want to get it from. In Pop11 the each instance variable has an associated accessor function, to get (or set) value of instance variable you pass the object as an argument to the accessor:

```
lvars blank_x = x(blank);
```

The expression `x(blank)` means "go to the object `blank` refers to, and get the value of `x`." In this case we assign that value to a variable named `blank_x`. We can not use the name `x` as the name of variable, because the name is already taken (by the accessor function).

In Pop11 you can also use "dot notation."

```
lvars blank_x = blank.x;
```

The expression `blank.x` is just a shorthand for `x(blank)`.

You can use dot notation as part of any Pop11 expression, so the following are legal.

```
blank.x ><  ', ' >< blank.y =>
lvars distance = blank.x * blank.x + blank.y * blank.y;
```

The first line prints `3, 4`; the second line calculates the value 25.

9.5 Objects as parameters

You can pass objects as parameters in the usual way. For example

```
define print_point (p) {
    printf ('(' >< x(p) >< ', ' >< y(p) + ')', '%s\n');
enddefine;
```

is a function that takes a point as an argument and prints it in the standard format. If you invoke `print_point (blank)`, it will print `(3, 4)`. Actually, Pop11 printf can also print `Points`. If you invoke `printf (blank, '%\n')`, you get

```
<Point x:3 y:4>
```

This is a standard format Pop11 uses for printing objects. It prints the name of the type, followed by the contents of the object, including the names and values of the instance variables.

As a second example, we can rewrite the `distance` function from Section 5.2 so that it takes two `Points` as parameters instead of four `doubles`.

```
define distance(p1, p2);
    lvars dx = x(p1) - x(p1);
    lvars dy = y(p2) - y(p1);
    return(sqrt(dx*dx + dy*dy));
enddefine;
```

9.6 Rectangles

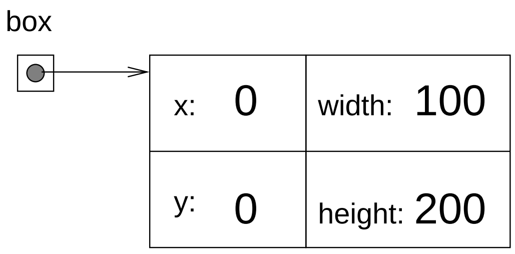
`Rectangles` are similar to points, except that they have four instance variables, named `x`, `y`, `width` and `height`.

Other than that, everything is pretty much the same:

```
define :class Rectangle;
   slot x = 0;
   slot y = 0;
   slot width = 0;
   slot height = 0;
enddefine;

lvars box = consRectangle (0, 0, 100, 200);
```

creates a new `Rectangle` object and makes `box` refer to it. The figure shows the effect of this assignment.



If you print `box`, you get

```
<Rectangle x:0 y:0 width:100 height:200>
```

Again, this result is because Pop11 has a built-in way to print all objects.

9.7 Objects as return types

You can write functions that return objects. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
define find_center (box);
    lvars x1 = x(box) + width(box)/2;
    lvars y1 = y(box) + height(box)/2;
    return(consPoint (x1, y1);
enddefine;
```

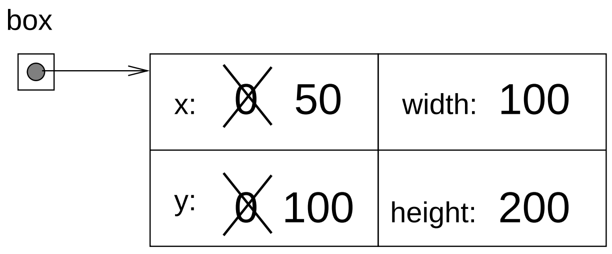
Notice that you can use `consPoint` to create a new object, and then immediately use the result as a return value.

9.8 Objects are mutable

You can change the contents of an object by making an assignment to one of its instance variables. For example, to "move" a rectangle without changing its size, you could modify the `x` and `y` values:

```
x(box) + 50 -> x(box);
y(box) + 100 -> y(box);
```

The result is shown in the figure:



We could take this code and encapsulate it in a function, and generalize it to move the rectangle by any amount:

```
define move_rect(box, dx, dy);
    x(box) + dx -> x(box);
    y(box) + dy -> y(box);
enddefine;
```

The variables `dx` and `dy` indicate how far to move the rectangle in each direction. Invoking this function has the effect of modifying the `Rectangle` that is passed as an argument.

```
lvars box = consRectangle (0, 0, 100, 200);
move_rect (box, 50, 100);
box =>
```

prints `<Rectangle x:50 y:100 width:100 height:200>`.

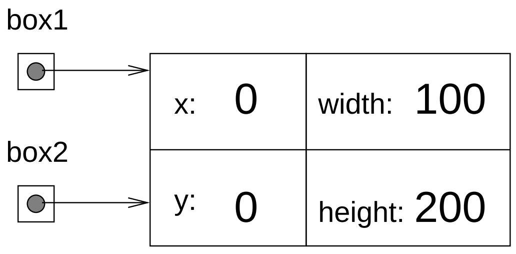
Modifying objects by passing them as arguments to functions can be useful, but it can also make debugging more difficult because it is not always clear which function invocations do or do not modify their arguments. Later, I will discuss some pros and cons of this programming style.

9.9 Aliasing

Remember that when you make an assignment to an object variable, you are assigning a *reference* to an object. It is possible to have multiple variables that refer to the same object. For example, this code:

```
lvars box1 = consRectangle (0, 0, 100, 200);
lvars box2 = box1;
```

generates a state diagram that looks like this:

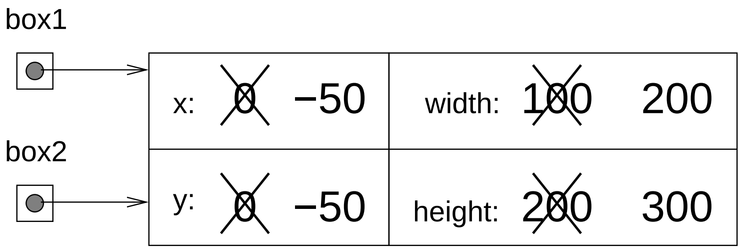


Both `box1` and `box2` refer or "point" to the same object. In other words, this object has two names, `box1` and `box2`. When a person uses two names, it's called **aliasing**. Same thing with objects.

When two variables are aliased, any changes that affect one variable also affect the other. For example:

```
box2.width =>
width(box1) + 50 -> width(box1);
height(box1) + 50 -> height(box1);
box2.width =>
```

The first line prints `100`, which is the width of the `Rectangle` referred to by `box2`. The second line and third line increase `width` and `height` of `box1` by 50 (in other word thos two lines expands the `box1` by 50 pixels in every direction). The effect is shown in the figure:



As should be clear from this figure, whatever changes are made to `box1` also apply to `box2`. Thus, the value printed by the third line is `200`, the width of the expanded rectangle. (As an aside, it is perfectly legal for the coordinates of a `Rectangle` to be negative.)

As you can tell even from this simple example, code that involves aliasing can get confusing fast, and it can be very difficult to debug. In general, aliasing should be avoided or used with care.

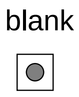
9.10 Empty list as a null object

When you create a variable, it is good to immediatly initialize it. In fact, if you do not provide inital value Pop11 will initialize the variable with special **undef** value (which serves as a marker to alert you that you are using an undefined value). However, somtimes the value in not really undefined, but just there in *no* value. For example a `Person` object may have an instance variable `spouse`: for a single person we want explicitly say that thare is no spouse. Some languages ifor this purpose use special **null** object. In Pop11 we can do this in multiple ways. One is to use a special value, for example a word like `"none"`. In the sequel we will use empty list (`[]`) to signify that a variable references no object.

For example

```
lvars blank = [];
```

is shown in the following state diagram:



The value [] is represented by a dot with no arrow.

If you try to use empty list in place of an object, either by accessing an instance variable or passing it as argument to a function expecting an object, you will get a mishap. The system will print an error message and terminate the program.

```
lvars blank = [];
lvars blank_x = x(blank);               ;;; mishap
move_rect(blank, 50, 50);      ;;; mishap
```

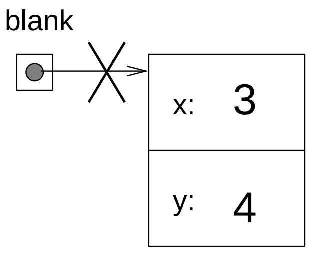
On the other hand, it is legal to pass a empty list as an argument or receive one as a return value. In fact, it is common to do so, for example to represent an empty set or indicate an error condition.

9.11 Garbage collection

In Section 9.9 we talked about what happens when more than one variable refers to the same object. What happens when *no* variable refers to an object? For example:

```
lvars blank = consPoint (3, 4);
[] -> blank;
```

The first line creates a new Point object and makes blank refer to it. The second line changes blank so that instead of referring to the object, it refers to the empty list.



If no one refers to an object, then no one can read or write any of its values, or invoke a function on it. In effect, it ceases to exist. We could keep the object in memory, but it would only waste space, so periodically as your program runs, the Pop11 system looks for stranded objects and reclaims them, in a process called **garbage collection**. Later, the memory space occupied by the object will be available to be used as part of a new object.

You don't have to do anything to make garbage collection work, and in general you will not be aware of it.

9.12 Objects and primitives

Strictly speaking Pop11 object types are differenent from other Pop11 types. However, Pop11 object system is so well integrated that the difference hardy matters.

9.13 Glossary

instance: An example from a category. My cat is an instance of the category "feline things." Every object is an instance of some class.

instance variable: One of the named data items that make up an object. Each object (instance) has its own copy of the instance variables for its class.

reference: A value that indicates an object. In a state diagram, a reference appears as an arrow.

aliasing: The condition when two or more variables refer to the same object.

garbage collection: The process of finding values that have no references and reclaiming their storage space.

state: A complete description of all the variables and objects and their values, at a given point during the execution of a program.

state diagram: A snapshot of the state of a program, shown graphically.

Chapter 10

More objects

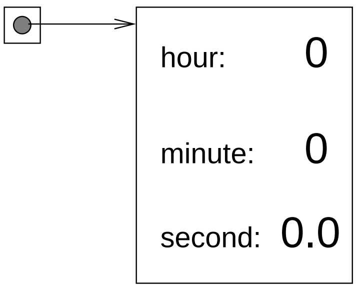
10.1 Time

Another example of class, is `Time`, which is used to record the time of day. The various pieces of information that form a time are the hour, minute and second. Because every `Time` object will contain these data, we need to create instance variables to hold them.

Instance variables are declared like this:

```
define :class Time;
    slot hour = 0;
    slot minute = 0;
    slot second = 0.0;
enddefine;
```

The state diagram for a `Time` object looks like this:



10.2 Constructors

The usual role of a constructor is to initialize the instance variables.

Given class definition of `Time` Pop11 automatically generates two constructor function `consTime` and `newTime`.

The `newTime` constructor does not take any arguments, it initializes instance variables to default values (either values given in the slot definition or special `undef` value).

The `consTime` constructor has a parameter list that is identical to the list of instance variables. All the constructor does is copy the information from the parameters to the instance variables.

If you go back and look at `Points` and `Rectangles`, you will see that both classes provide constructors like this (we used only `consPoint` and `consRectangle`). Have both constructors provides the flexibility to create an object first and then fill in the blanks, or to collect all the information before creating the object.

Of course, sometimes the two automatically generated constructors are not enough. For example, we want a constructor that has inital values for some instance variables as parameters, and based on them computes to other values. Such constructor is easy to write: just use the `new` variant of standard constructor to a blank object, and fill the instance variables with required values.

10.3 Creating a new object

Constructors are just ordinary function, you invoke them like all other functions.

The following program demonstrates two ways to create and initialize `Time` objects:

```
;;; one way to create and initialize a Time object
lvars t1 = newTime ();
11 -> hour(t1);
8 -> minute(t1);
3.14159 -> second(t1);
t1 =>

;;; another way to do the same thing
lvars t2 = consTime (11, 8, 3.14159);
t2 =>
```

The first time we invoke the `newTime` function, it takes no arguments. The next few lines assign values to each of the instance variables.

The second time we invoke the `consTime` function, we provide values of instance variables as arguments. This way of initializing the instance variables is more concise (and slightly more efficient), but it can be harder to read, since it is not as clear which values are assigned to which instance variables.

10.4 Printing an object

The output of this program is:

```
** <Time hour:11 minute:8 second:3.14159>
** <Time hour:11 minute:8 second:3.14159>
```

When Pop11 prints the value of a user-defined object type, it prints the name of the type and names and values of all instance variables, surrounde by `<` and `>`. The name of instance variables is separated from value by colon (`:`).

In order to print objects in a way that is more meaningful to users (as opposed to programmers), you may want to write a function called something like `print_time`:

```
define print_time (t);
    printf (hour(t) >< ':' >< minute(t) >< ':' >< second(t),
            '%p\n');
enddefine;
```

Compare this function to the version of `print_time` in Section 3.8.

The output of this function, if we pass either `t1` or `t2` as an argument, is `11:8:3.14159`. Although this is recognizable as a time, it is not quite in the standard format. For example, if the number of minutes or seconds is less than 10, we expect a leading `0` as a place-keeper. Also, we might want to drop the decimal part of the seconds. In other words, we want something like `11:08:03`.

Pop11 provides very powerful `format_print` function for printing formatted things like numbers. We will get nicer output if we replace the `printf` invocation in `print_time` by the following:

```
    format_print('~A:~A:~7,4,,,`0F\n',
                 [^(hour(t)) ^(minute(t)) ^(second(t))]);
```

10.5 Operations on objects

Beside printing, we can write other function that manipulate `Time` objects. In the next few sections, I will demonstrate several functions that operate on objects. For some operations, you will have a choice of several possible styles, so you should consider the pros and cons of each of these:

pure function: Takes objects and/or primitives as arguments but does not modify the objects. The return value is either a primitive or a new object created inside the function.

modifier: Takes objects as arguments and modifies some or all of them. Often returns no value.

fill-in function: One of the arguments is an "empty" object that gets filled in by the function. Technically, this is a type of modifier.

10.6 Pure functions

A function is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or printing something. The only result of invoking a pure function is the return value.

One example is `after`, which compares two `Times` and returns a `boolean` that indicates whether the first operand comes after the second:

```
define after(time1,time2);
    if hour(time1) > hour(time2) then
        return(true);
    elseif hour(time1) < hour(time2) then
        return(false);
    elseif minute(time1) > minute(time2) then
        return(true);
    elseif minute(time1) < minute(time2) then
        return(false);
    elseif second(time1) > second(time2) then
        return(true);
    elseif second(time1) < second(time2) then
        return(false);
    endif;
    return(false);
enddefine;
```

What is the result of this function if the two times are equal? Does that seem like the appropriate result for this function? If you were writing the documentation for this function, would you mention that case specifically?

A second example is `add_time`, which calculates the sum of two times. For example, if it is `9:14:30`, and your breadmaker takes 3 hours and 35 minutes, you could use `add_time` to figure out when the bread will be done.

Here is a rough draft of this function that is not quite right:

```
define add_time(t1, t2);
    lvars sum = newTime();
    hour(t1) + hour(t2) -> hour(sum);
    minute(t1) + minute(t2)-> minute(sum);
    second(t1) + second(t2) -> second(sum);
    return(sum);
enddefine;
```

Here is an example of how to use this function. If `current_time` contains the current time and `bread_time` contains the amount of time it takes for your breadmaker to make bread, then you could use `add_time` to figure out when the bread will be done.

```
lvars current_time = consTime (9, 14, 30.0);
lvars bread_time = consTime (3, 35, 0.0);
lvars done_time = add_time (current_time, bread_time);
print_time (done_time);
```

The output of this program is `12:49:30.0`, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than 60. In that case, we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a second, corrected version of this function.

```
define add_time(t1, t2);
    lvars sum = newTime();
    hour(t1) + hour(t2) -> hour(sum);
    minute(t1) + minute(t2)-> minute(sum);
    second(t1) + second(t2) -> second(sum);
    if second(sum) >= 60.0 then
        second(sum) - 60.0 -> second(sum);
        minute(sum) + 1 -> minute(sum);
    endif;
    if minute(sum) >= 60 then
        minute(sum) - 60 -> minute(sum);
        hour(sum) + 1 -> hour(sum);
    endif;
    return(sum);
enddefine;
```

Although it's correct, it's starting to get big. Later, I will suggest an alternate approach to this problem that will be much shorter.

10.7 Modifiers

As an example of a modifier, consider `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this function looks like:

```
define increment(time, secs);
    second(time) + secs -> second(time);
    if second(time) >= 60.0 then
        second(time) - 60.0 -> second(time);
        minute(time) + 1 -> minute(time);
    endif;
    if minute(time) >= 60 then
        minute(time) - 60 -> minute(time);
        hour(time) + 1 -> hour(time);
    endif;
enddefine;
```

The first line performs the basic operation; the remainder deals with the same cases we saw before.

Is this function correct? What happens if the argument `secs` is much greater than 60? In that case, it is not enough to subtract 60 once; we have to keep doing it until `second` is below 60. We can do that by simply replacing the `if` statements with `while` statements:

```
define increment(time, secs);
    second(time) + secs -> second(time);
    while second(time) >= 60.0 then
        second(time) - 60.0 -> second(time);
        minute(time) + 1 -> minute(time);
    endif;
    while minute(time) >= 60 then
        minute(time) - 60 -> minute(time);
        hour(time) + 1 -> hour(time);
    endif;
enddefine;
```

This solution is correct, but not very efficient. Can you think of a solution that does not require iteration?

10.8 Fill-in functions

Occasionally you will see functions like `add_time` written with a different interface (different arguments and return values). Instead of creating a new object every time `add_time` is invoked, we could require the caller to provide an "empty" object where `add_time` should store the result. Compare the following with the previous version:

```
define add_time_fill(t1, t2, sum);
    hour(t1) + hour(t2) -> hour(sum);
    minute(t1) + minute(t2)-> minute(sum);
    second(t1) + second(t2) -> second(sum);
    if second(sum) >= 60.0 then
        second(sum) - 60.0 -> second(sum);
        minute(sum) + 1 -> minute(sum);
    endif;
    if minute(sum) >= 60 then
        minute(sum) - 60 -> minute(sum);
        hour(sum) + 1 -> hour(sum);
    endif;
enddefine;
```

One advantage of this approach is that the caller has the option of reusing the same object repeatedly to perform a series of additions. This can be slightly more efficient, although it can be confusing enough to cause subtle errors. For the vast majority of programming, it is worth spending a little run time to avoid a lot of debugging time.

10.9 Which is best?

Anything that can be done with modifiers and fill-in functions can also be done with pure functions. In fact, there are programming languages, called **functional** programming languages, that only allow pure functions. Some programmers believe that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, there are times when modifiers are convenient, and some cases where functional programs are less efficient.

In general, I recommend that you write pure functions whenever it is reasonable to do so, and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

10.10 Incremental development vs. planning

In this chapter I have demonstrated an approach to program development I refer to as **rapid prototyping with iterative improvement**. In each case, I wrote a rough draft (or prototype) that performed the basic calculation, and then tested it on a few cases, correcting flaws as I found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to convince yourself that you have found *all* the errors.

An alternative is high-level planning, in which a little insight into the problem can make the programming much easier. In this case the insight is that a `Time` is really a three-digit number in base 60! The `second` is the "ones column," the `minute` is the "60's column", and the `hour` is the "3600's column."

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Thus an alternate approach to the whole problem is to convert `Time`s into a floating point value and take advantage of the fact that the computer already knows how to do arithmetic with floating point values. Here is a function that converts a `Time` into a `ddecimal`:

```
define convert_to_seconds (t) {
    lvars minutes = hour(t) * 60 + minute(minute);
    lvars seconds = minutes * 60 + second(t);
    return(seconds);
enddefine;
```

Now all we need is a way to convert from a `ddecimal` to a `Time` object: do it, but it might make more sense to write it as a third constructor:

```
define seconds_to_time(secs)
    lvars this = newTime();
    intof(secs / 3600.0) -> hour(this);
    secs - hour(this) * 3600.0 -> secs;
    intof(secs / 60.0) -> minute(this);
    secs - minute(this) * 60 -> secs;
    second -> secs second(this);
    return(this);
enddefine;
```

This function is really a third constructor for `Time`. This constructor is a little different from the others, since it involves some calculation along with assignments to the instance variables.

It is quite tricky why the technique I am using to convert from one base to another is correct (assuming that `secs` is non-negative). With some effort you should see that this works *provided that the arithmetic operations are done exactly.* However, floating point operation give only approximate result (due to rounding). Fortunatly, on modern machines the `intof` operation is exact. Also, integers of reasonable size (in particular `3600.0` are represented exactly. Next, if x is an exact integer multiple of n and $y > x$ then floating point division of y by n gives bigger result than floating point division of x by n. Similarly, if x is an exact integer multiple of n and $y < x$ then loating point division of y by n gives smaller result than floating point division of x by n. The net effect is that inequalities that matter for us (for example that `second(this)` is smaller than 60) hold the same as in case of exact calculations.

Assuming you belive that `seconds_to_time` is correct, we can use these functionss to rewrite `add_time`:

```
define add_time(t1, t2);
    lvars seconds = conver_to_seconds (t1) + convert_to_seconds (t2);
    return(seconds_to_time(seconds));
enddefine;
```

This is much shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the functions it invokes are correct). As an exercise, rewrite `increment` the same way.

10.11 Generalization

In some ways converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers, and make the investment of writing the conversion functions (`convert_to_seconds` and `seconds_to_time`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add more features later. For example, imagine subtracting two `Times` to find the duration between them. The naive approach would be to implement subtraction complete with "borrowing." Using the conversion functions would be much easier.

Ironically, sometimes making a problem harder (more general) makes is easier (fewer special cases, fewer opportunities for error).

10.12 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. I mentioned this word in Chapter 1, but did not define it carefully. It is not easy to define, so I will try a couple of approaches.

First, consider some things that are not algorithms. For example, when you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions, so that knowledge is not really 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 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 most difficult 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.

Later you will have the opportunity to design simple algorithms for a variety of problems.

10.13 Glossary

class: In this chapter we learned that a class definition is a template for a new type of object.

instance: A member of a class. Every object is an instance of some class.

constructor: A special function that creates a new object and initializes its instance variables.

pure function: A function whose result depends only on its parameters, and that has no side-effects other than returning a value.

functional programming style: A style of program design in which the majority of functions are pure.

modifier: A function that changes one or more of the objects it receives as parameters, and usually returns no value.

fill-in function: A type of function that takes an "empty" object as a parameter and fills it its instance variables instead of generating a return value. This type of function is usually not the best choice.

algorithm: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

Chapter 11

Arrays

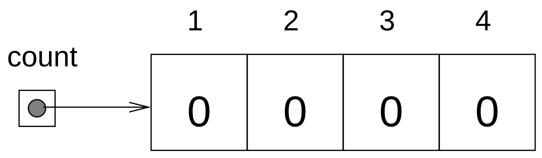
An **array** is a set of values where each value is identified by an index. Pop11 has to kinds of arrays, a simple one called `vector` and more involved one called `array`. Most of the time `vector`s will suffice, so talking about arrays I will mostly talk about `vector`s and only rarely about general `array`s.

The simplest way to create a vector is to use curly braces {}

```
lvars count = {0 0 0 0}
```

The first initialization makes `count` refer to a fresh vector containing four integer zeros.

The following figure shows how arrays are represented in state diagrams:



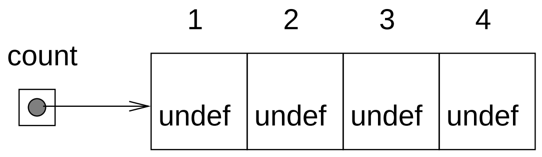
The words inside the boxes are the **elements** of the array. The small numbers outside the boxes are the indices used to identify each box.

To create a vector, you can also use the `initv` function.

```
lvars count = initv(4);
```

This initialization makes `count` refer to a fresh four element vector filled with the word `undef`. We gave no initial value for elements, so Pop11 uses the word `undef` to remaind us to initialize the elements with correct values.

The following figure shows the new array:

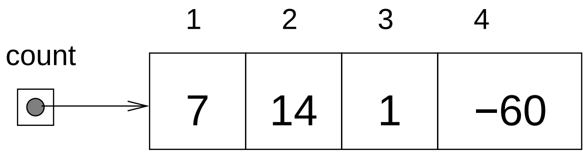


11.1 Accessing elements

To store values in the array, use the `()` operator. For example `count(1)` refers to the first element of the array, and `count(2)` refers to the second element. You can use the `()` operator anywhere in an expression:

```
7 -> count(1);
count(1) * 2 -> count(2);
-(count(1) rem 2) -> count(3);
-59 + count(3) -> count(4);
```

All of these are legal assignment statements. Here is the effect of this code fragment:



By now you should have noticed that the four elements of this array are numbered from 1 to 4, which means that there is no element with the index 5. This should sound familiar, since we saw the same thing with string indices. Nevertheless, it is a common error to go beyond the bounds of an array, which will cause an exception (mishap). As with all exceptions, you get an error message and the program quits.

You can use any expression as an index, as long as its values are integers. One of the most common ways to index an array is with a loop variable. For example:

```
lvars i = 1;
while i <= 4 do
    printf(count(i), '%p\n');
    i + 1 -> i;
endwhile;
```

This is a standard `while` loop that counts from 1 up to 4, and when the loop variable `i` is 5, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 1, 2, 3 and 4.

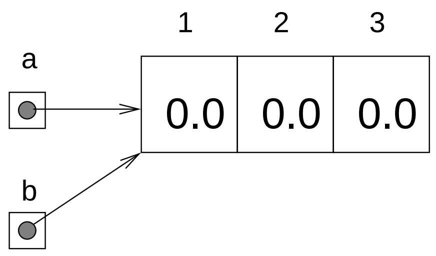
Each time through the loop we use `i` as an index into the array, printing the `i`th element. This type of array traversal is very common. Arrays and loops go together like fava beans and a nice Chianti.

11.2 Copying arrays

When you copy an array variable, remember that you are copying a reference to the array. For example:

```
lvars a = { 0.0 0.0 0.0 };
lvars b = a;
```

This code creates one array of three `decimals`, and sets two different variables to refer to it. This situation is a form of aliasing.



Any changes in either array will be reflected in the other. This is not usually the behavior you want; instead, you should make a copy of the array. You can use `copy` function

```
lvars b = copy(a);
```

Alternatively, you can allocate a new array and copy each element from one to the other.

```
lvars b = initv(3);

lvars i = 1;
while i <= 3 do
    a(i) -> b(i);
    i + 1 -> i;
endwhile;
```

11.3 `for` loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they increment that variable.

This type of loop is so common that there is a variant of `for` statement, that expresses it more concisely. The general syntax looks like this:

```
for var from START by STEP to END do
    BODY
endfor;
```

If STEP is positive this statement is exactly equivalent to

```
START -> var;
while var <= END do
```

```
    BODY
    STEP + var -> var;
endwhile;
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. If STEP is 1 you can omit the `by STEP` part. For example:

```
lvars i;
for i from 1 to 4 do
    printf (count(i), '%p\n');
endfor;
```

is equivalent to

```
lvars i = 1;
while i <= 4 do
    printf (count(i), '%p\n');
    i + 1 -> i;
endwhile;
```

As an exercise, write a `for` loop to copy the elements of an array.

11.4 Arrays and objects

In many ways, arrays behave like objects:

- When you declare an array variable, you get a reference to an array.
- You have to use the `new` command to create the array itself.
- When you pass an array as an argument, you pass a reference, which means that the invoked function can change the contents of the array.

Some of the objects we have looked at, like `Rectangles`, are similar to arrays, in the sense that they are named collection of values. This raises the question, "How is an array of 4 integers different from a Rectangle object?"

If you go back to the definition of "array" at the beginning of the chapter, you will see one difference, which is that the elements of an array are identified by indices, whereas the elements (instance variables) of an object have names (like `x`, `width`, etc.).

11.5 Array length

The `length` function works for arrays (the length of an array is the number of elements). It is a good idea to use this value as the upper bound of a loop, rather than a constant value. That way, if the size of the array changes, you won't have to go through the program changing all the loops; they will work correctly for any size array.

```
for i from 1 to length(a) do
    a(i) -> b(i);
endfor;
```

The last time the body of the loop gets executed, `i` is `length(a)`, which is the index of the last element. This code assumes that the array `b` contains at least as many elements as `a`.

As an exercise, write your own version of `copy` function for `vectors` (call it `clone_array`) that takes a vector as a parameter, creates a new vector that is the same size, copies the elements from the first array into the new one, and then returns a reference to the new array.

11.6 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example, but there are many 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 generate random numbers and use them to determine the outcome of the program. Pop11 provides a built-in function that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes, they will do.

Check out the documentation of the `random` function. The `random` function takes one argument, an integer or a float. The return value is a `integer` bigger than 0 and smaller or equal to the argument. If the argument is an integer the return value is also an integer, if the argument is a float return value is a float. Each time you invoke `random` you get a different randomly-generated number. To see a sample, run this loop:

```
lvars i;
for i from 1 to 10 do
     lvars x = random (1.0);
     printf (x, '%p\n');
endfor;
```

To generate a random float between `low` and `high high`, you can add `low` to `x`. Try to generate a random integer.

11.7 Statistics

The numbers generated by `random` are supposed to be distributed uniformly. If you have taken statistics, you know what that means. Among other things, it means that if we divide the range of possible values into equal sized "buckets," and count the number of times a random value falls in each bucket, each bucket should get the same number of hits (eventually).

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

11.8 Array of random numbers

The first step is to generate a large number of random values and store them in an array. By "large number," of course, I mean 8. It's always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following function takes a single argument, the size of the array. It allocates a new array, fills it with random floats, and returns a reference to the new array.

```
define random_array(n);
    lvars a = initv(n);
    lvars i;
    for i from 1 to length(a) do
        random(1.0) -> a(i);
    endfor;
    return (a);
enddefine;
```

To test this function, it is convenient to print the contents of an array – print arrow works fine for vectors.

The following code generates an array and prints it:

```
lvars num_values = 8;
lvars array = random_array (num_values);
array =>
```

On my machine the output is

```
** {0.511548 0.465616 0.1745 0.714364 0.086424 0.390443 0.962521 0.468249}
```

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect half of them to be greater than 0.5 and half to be less. In fact, five are smaller than 0.5, so that's a little different.

If we divide the range into four buckets—from 0.0 to 0.25, 0.25 to 0.5, 0.5 to 0.75, and 0.75 to 1.0—we expect 2 values to fall in each bucket. In fact, we get 2, 3, 2, 1. Again, not exactly what we expected.

Do these results mean the values are not really random? It's hard to tell. With so few values, the chances are slim that we would get exactly what we expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that divide the range into buckets and count the number of values in each.

11.9 Counting

A good approach to problems like this is to think of simple functions that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. Of course, it is not easy to know ahead of time which functions are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

Back in Section 7.8 we looked at a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like an array or a string.
- A test that you can apply to each element in the container.
- A counter that keeps track of how many elements pass the test.

In this case, I have a function in mind called `in_bucket` that counts the number of elements in an array that fall in a given bucket. The parameters are the array and two floats that specify the lower and upper bounds of the bucket.

```
define in_bucket(a, low, high);
    lvars count = 0;
    lvars i;
    for i from 1 to length(a) do
        if a(i) > low and a(i) <= high then
            count + 1 -> count;
        endif;
    endfor;
    return (count);
enddefine;
```

I haven't been very careful about whether something equal to `low` or `high` falls in the bucket, but you can see from the code that `low` is out and `high` is in. That should prevent me from counting any elements twice.

Now, to divide the range into two pieces, we could write

```
lvars low = in_bucket (a, 0.0, 0.5);
lvars high = in_bucket (a, 0.5, 1.0);
```

To divide it into four pieces:

```
lvars bucket1 = in_bucket (a, 0.0, 0.25);
lvars bucket2 = in_bucket (a, 0.25, 0.5);
lvars bucket3 = in_bucket (a, 0.5, 0.75);
lvars bucket4 = in_bucket (a, 0.75, 1.0);
```

You might want to try out this program using a larger `num_values`. As `num_values` increases, are the numbers in each bucket levelling off?

11.10 Many buckets

Of course, as the number of buckets increases, we don't want to have to rewrite the program, especially since the code is getting big and repetitive. Any time you find yourself doing something more than a few times, you should be looking for a way to automate it.

Let's say that we wanted 8 buckets. The width of each bucket would be one eighth of the range, which is 0.125. To count the number of values in each bucket, we need to be able to generate the bounds of each bucket automatically, and we need to have some place to store the 8 counts.

We can solve the first problem with a loop:

```
lvars num_buckets = 8;
lvars bucket_width = 1.0 / num_buckets;

for i from 0 to num_buckets - 1 do
    lvars low = i * bucket_width;
    lvars high = low + bucket_width;
    printf(low >< ' to ' >< high, '%p\n');
endfor;
```

This code uses the loop variable `i` to multiply by the bucket width, in order to find the low end of each bucket. The output of this loop is:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

You can confirm that each bucket is the same width, that they don't overlap, and that they cover the whole range from 0.0 to 1.0.

Now we just need a way to store 8 integers, preferably so we can use an index to access each one. Immediately, you should be thinking "array!"

What we want is an array of 8 integers, which we can allocate outside the loop; then, inside the loop, we'll invoke `in_bucket` and store the result:

```
lvars num_buckets = 8;
lvars buckets = initv(num_buckets);
lvars bucket_width = 1.0 / num_buckets;

for i from 0 to num_buckets - 1 do
    lvars low = i * bucket_width;
    lvars high = low + bucket_width;
    ;;; printf(low >< ' to ' >< high, '%p\n');
    in_bucket (a, low, high) -> buckets(i + 1);
endfor;
```

The tricky thing here is that I am using the loop variable as an index into the `buckets` array, in addition to using it to compute the range of each bucket.

This code works. I cranked the number of values up to 1000 and divided the range into 8 buckets. The output is:

```
111 146 136 107 135 117 122 126
```

which is pretty close to 125 in each bucket. At least, it's close enough that I can believe the random number generator is working.

11.11 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it invokes `in_bucket`, it traverses the entire array. As the number of buckets increases, that gets to be a lot of traversals.

It would be better to make a single pass through the array, and for each value, compute which bucket it falls in. Then we could increment the appropriate counter.

In the previous section, we took an index, `i`, and multiplied it by the `bucket_width` in order to find the lower bound of a given bucket. Now we want to take a value in the range 0.0 to 1.0, and find the index of the bucket where it falls.

Since this problem is the inverse of the previous problem we might guess that we should *divide* by the `bucket_width` instead of multiplying. That guess is correct.

Remember that since `bucket_width = 1.0 / num_buckets`, dividing by `bucket_width` is the same as multiplying by `num_buckets`. If we take a number in the range 0.0 to 1.0 and multiply by `num_buckets`, we get a number in the range from 0.0 to `num_buckets`. If we round that number to the next lower integer and add 1, we get exactly what we are looking for—the index of the appropriate bucket.

```
lvars num_buckets = 8;
lvars buckets = initv(num_buckets);
lvars i;

for i from 1 to num_buckets do
    0 -> buckets(i);
endfor;

for i from 1 to num_values do
    lvars index = 1 + intof(a(i) * num_buckets);
    buckets(index) + 1 -> buckets(index);
endfor;
```

Here I am using the `intof` to round the value down to the next integer and convert it to integer at the same time.

Is it possible for this calculation to produce an index that is out of range (either negative or greater than `length(a)`)? If so, how would you fix it?

An array like `buckets`, that contains counts of the number of values in each range, is called a **histogram**. As an exercise, write a function called `histogram` that takes an array and a number of buckets as parameters, and that returns a histogram with the given number of buckets.

11.12 Glossary

array: A named collection of values, where each value is identified by an index.

collection: Any data structure that contains a set of items or elements.

element: One of the values in an array. The `()` operator selects elements of an array.

index: An integer variable or value used to indicate an element of an array.

deterministic: A program that does the same thing every time it is invoked.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

Chapter 12

Arrays of Objects

12.1 Composition

By now we have seen several examples of composition (the ability to combine language features in a variety of arrangements). One of the first examples we saw was using a function invocation as part of an expression. Another example is the nested structure of statements: you can put an `if` statement within a `while` loop, or within another `if` statement, etc.

Having seen this pattern, and having learned about arrays and objects, you should not be surprised to learn that you can have arrays of objects. In fact, you can also have objects that contain arrays (as instance variables); you can have arrays that contain arrays; you can have objects that contain objects, and so on.

In the next two chapters we will look at some examples of these combinations, using `Card` objects as an example.

12.2 `Card` objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are 52 cards in a deck, each of which belongs to one of four suits and one of 13 ranks. The suits are Spades, Hearts, Diamonds and Clubs (in descending order in Bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King. Depending on what game you are playing, the rank of the Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is pretty obvious what the instance variables should be: `rank` and `suit`. It is not as obvious what values to store in the instance variables. One possibility is words, usings things like `"Spade"` for suits and `"Queen"` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By “encode,” I do not mean what some people think, which is to encrypt, or translate into a secret code. What a computer scientist means by “encode” is something like “define a mapping between a sequence of numbers and the things I want to represent.” For example,

Spades	$\mapsto$	4
Hearts	$\mapsto$	3
Diamonds	$\mapsto$	2
Clubs	$\mapsto$	1

The symbol $\mapsto$ is mathematical notation for “maps to.” The obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

The reason I am using mathematical notation for these mappings is that they are not part of the Pop11 program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```
define :class Card;
   slot suit = 1;
   slot rank = 1;
enddefine;
```

As usual, we get two constructors, one of which takes a parameter for each instance variable and the other of which takes no parameters.

To create an object that represents the 3 of Clubs, we would use the `consCard` command:

```
lvars threeOfClubs = consCard (1, 3);
```

The first argument, `1` represents the suit Clubs.

12.3 The `print_card` function

When you create a new class, the first step is usually to declare the instance variables and write constructors. The second step is often to write the standard functions that every object should have, including one that prints the object, and one or two that compare objects. I will start with `print_card`.

In order to print `Card` objects in a way that humans can read easily, we want to map the integer codes onto words. A natural way to do that is with an array of words. You can create an array of words the same way you create an array of other primitive types:

```
lvars suits = initv(4);
```

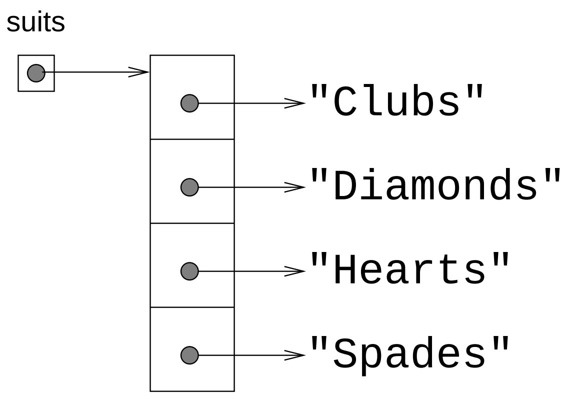
Then we can set the values of the elements of the array.

```
"Clubs" -> suits(1);
"Diamonds" -> suits(2);
"Hearts" -> suits(3);
"Spades" -> suits(4);
```

Creating an array and initializing the elements is such a common operation that Pop11 provides a special syntax for it:

```
lvars suits = { Clubs Diamonds Hearts Spades };
```

The effect of this statement is identical to that of the separate declaration, allocation, and assignment. A state diagram of this array might look like:



Now, all we need is another array to decode the ranks:

```
lvars ranks = { Ace 2 3 4 5 6 7 8 9 10 Jack Queen King };
```

This time, for convenience, we mixed words with integers.

Using these arrays, we can select the appropriate value to print by using the `suit` and `rank` as indices. In the function `print_card`,

```
define print_card(c);
    lvars suits = { Clubs Diamonds Hearts Spades };
    lvars ranks = { Ace 2 3 4 5 6 7 8 9 10 Jack Queen King };
    printf(ranks(rank(c)) >< ' of ' >< suits(suit(c)), '%p\n');
enddefine;
```

the expression `suits(suit(c))` means "use the instance variable `suit` from the object `c` as an index into the array named `suits`, and select the appropriate value." The output of this code

```
lvars card = consCard (2, 11);
print_card (card);
```

is `Jack of Diamonds`.

12.4 Comparing `Cards`

The word “same” is one of those things that occur in natural language that seem perfectly clear until you give it some thought, and then you realize there is more to it than you expected.

For example, if I say “Chris and I have the same car,” I mean that his car and mine are the same make and model, but they are two different cars. If I say “Chris and I have the same mother,” I mean that his mother and mine are one and the same. So the idea of “sameness” is different depending on the context.

When you talk about objects, there is a similar ambiguity. For example, if two `Card`s are the same, does that mean they contain the same data (rank and suit), or they are actually the same `Card` object?

To see if two references refer to the same object, we can use the `==` operator. For example:

```
lvars card1 = consCard (1, 11);
lvars card2 = card1;
if card1 == card2 then
    printf('card1 and card2 are the same object.\n');
endif;
```

This type of equality is called **shallow equality** because it only compares the references, not the contents of the objects.

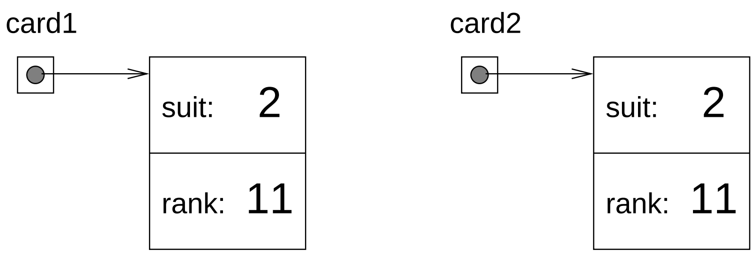
To compare the contents of the objects—**deep equality**— we use `=` operator.

Now, we create two different objects that contain the same data, we can use `=` to see if they represent the same card:

```
lvars card1 = consCard (2, 11);
lvars card2 = consCard (2, 11);

if card1 = card2 then
    printf('card1 and card2 are the same card.\n');
endif;
```

In this case, `card1` and `card2` are two different objects that contain the same data



so the condition is true. What does the state diagram look like when `card1 == card2` is true?

In Section 7.13 I said that you should never use the `==` operator on strings because it does not do what you expect. Instead of comparing the contents of the stings (deep equality), it checks whether the two strings are the same object (shallow equality).

12.5 The `compare_card` function

For (real) numbers, there are conditional operators that compare values and determine when one is greater or less than another. These operators (`<` and `>` and the others) don't work for other types. For stringss there is a built-in `alphabefore` function. For `Card`s we have to write our own, which we will call `compare_card`. Later, we will use this function to sort a deck of cards.

Some sets are completely ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are totally ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why we cannot compare apples and oranges. In Pop11, the `boolean` type is unordered; we cannot say that `true` is greater than `false`.

The set of playing cards is partially ordered, which means that sometimes we can compare cards and sometimes not. For example, I know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, we have to decide which is more important, rank or suit. To be honest, the choice is completely arbitrary. For the sake of choosing, I will say that suit is more important, because when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `compare_card`. It will take two `Card`s as parameters and return 1 if the first card wins, -1 if the second card wins, and 0 if they tie (indicating deep equality). It is sometimes confusing to keep those return values straight, but they are pretty standard for comparison functions.

First we compare the suits:

```
if suit(c1) > suit(c2) then
    return (1);
endif;
if suit(c1) < suit(c2) then
    return (-1);
endif;
```

If neither statement is true, then the suits must be equal, and we have to compare ranks:

```
if rank(c1) > rank(c2) then
```

```
    return(1);
endif;
if rank(c1) < rank(c2) then
    return(-1);
endif;
```

If neither of these is true, the ranks must be equal, so we return `0`. In this ordering, aces will appear lower than deuces (2s).

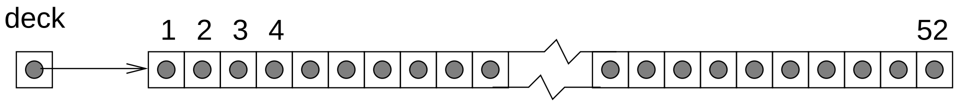
As an exercise, fix it so that aces are ranked higher than Kings, and encapsulate this code in a function.

12.6 Arrays of cards

The reason I chose `Cards` as the objects for this chapter is that there is an obvious use for an array of cards—a deck. Here is some code that creates a new deck of 52 cards:

```
lvars deck = initv(52);
```

Here is the state diagram for this object:



The important thing to see here is that the array contains only *references* to objects; it does not contain any `Card` objects. The values of the array elements are initialized to the word `undef`. You can access the elements of the array in the usual way:

```
if deck(3) == "undef" then
    printf ('No cards yet!\n');
endif;
```

But if you try to access the instance variables of the non-existent `Cards`, you will get a `Method "rank" failed` mishap.

```
rank(deck(3));              ;;; Method "rank" failed
```

Nevertheless, that is the correct syntax for accessing the `rank` of the third card in the deck. This is another example of composition, the combination of the syntax for accessing an element of an array and an instance variable of an object.

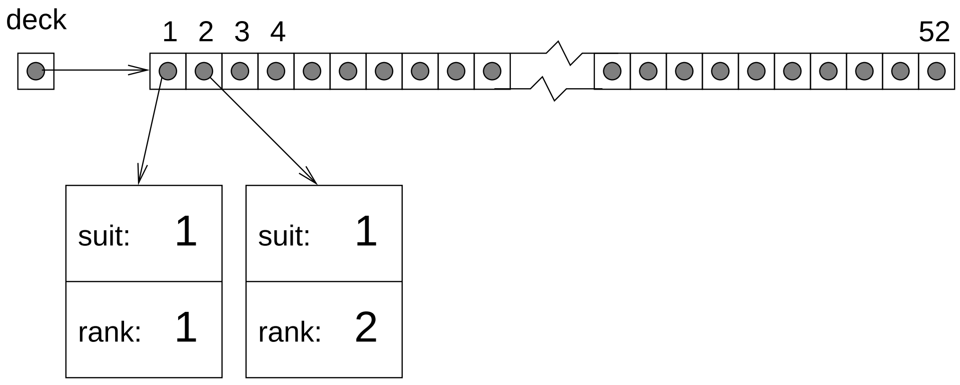
The easiest way to populate the deck with `Card` objects is to write a nested loop:

```
lvars index = 1;
lvars suit, rank;
```

```
for suit from 1 to 4 do
    for rank from 1 to 13 do
        consCard(suit, rank) -> deck(index);
        index + 1 -> index;
    endfor;
endfor;
```

The outer loop enumerates the suits, from 1 to 4. For each suit, the inner loop enumerates the ranks, from 1 to 13. Since the outer loop iterates 4 times, and the inner loop iterates 13 times, the total number of times the body is executed is 52 (13 times 4).

I used the variable `index` to keep track of where in the deck the next card should go. The following state diagram shows what the deck looks like after the first two cards have been allocated:



As an exercise, encapsulate this deck-building code in a function called `build_deck` that takes no parameters and that returns a fully-populated array of `Cards`.

12.7 The `print_deck` function

Whenever you are working with arrays, it is convenient to have a function that will print the contents of the array. We have seen the pattern for traversing an array several times, so the following function should be familiar:

```
define print_deck(deck);
    lvars i;
    for i from 1 to length(deck) do
       print_card(deck(i));
    endfor;
enddefine;
```

12.8 Searching

The next function I want to write is `find_card`, which searches through an array of `Card`s to see whether it contains a certain card. It may not be obvious why this function would be useful, but it gives me a chance to demonstrate two ways to go searching for things, a `linear` search and a `bisection` search.

Linear search is the more obvious of the two; it involves traversing the deck and comparing each card to the one we are looking for. If we find it we return the index where the card appears. If it is not in the deck, we return -1.

```
define find_card (deck, card);
    lvars i;
    for i from 1 to length(deck) do
        if deck(i) = card then
            return(i);
        endif;
    endfor;
    return -1;
enddefine;
```

The arguments of `find_card` are named `card` and `deck`. It might seem odd to have a variable with the same name as a type (the `card` variable has type `Card`). This is legal and common, although it can sometimes make code hard to read. In this case, though, I think it works.

The function returns as soon as it discovers the card, which means that we do not have to traverse the entire deck if we find the card we are looking for. If the loop terminates without finding the card, we know the card is not in the deck and return `-1`.

If the cards in the deck are not in order, there is no way to search that is faster than this. We have to look at every card, since otherwise there is no way to be certain the card we want is not there.

But when you look for a word in a dictionary, you don't search linearly through every word. The reason is that the words are in alphabetical order. As a result, you probably use an algorithm that is similar to a bisection search:

1. Start in the middle somewhere.
2. Choose a word on the page and compare it to the word you are looking for.
3. If you found the word you are looking for, stop.
4. If the word you are looking for comes after the word on the page, flip to somewhere later in the dictionary and go to step 2.
5. If the word you are looking for comes before the word on the page, flip to somewhere earlier in the dictionary and go to step 2.

If you ever get to the point where there are two adjacent words on the page and your word comes between them, you can conclude that your word is not in the dictionary. The only alternative is that your word has been misfiled somewhere, but that contradicts our assumption that the words are in alphabetical order.

In the case of a deck of cards, if we know that the cards are in order, we can write a version of `find_card` that is much faster. The best way to write a bisection search is with a recursive function. That's because bisection is naturally recursive.

The trick is to write a function called `find_bisect` that takes two indices as parameters, `low` and `high`, indicating the segment of the array that should be searched (including both `low` and `high`).

1. To search the array, choose an index between `low` and `high` (call it `mid`) and compare it to the card you are looking for.

2. If you found it, stop.

3. If the card at `mid` is higher than your card, search in the range from `low` to `mid-1`.

4. If the card at `mid` is lower than your card, search in the range from `mid+1` to `high`.

Steps 3 and 4 look suspiciously like recursive invocations. Here's what this all looks like translated into Pop11 code:

```
define find_bisect(deck, card, low, high);
    lvars mid = (high + low) div 2;
    lvars comp = compare_card (deck(mid), card);

    if comp = 0 then
        return(mid);
    elseif comp > 0 then
        return(find_bisect (deck, card, low, mid-1));
    else
        return(find_bisect (deck, card, mid+1, high));
    endif;
enddefine;
```

Rather than call `compare_card` three times, I called it once and stored the result.

Although this code contains the kernel of a bisection search, it is still missing a piece. As it is currently written, if the card is not in the deck, it will recurse forever. We need a way to detect this condition and deal with it properly (by returning `-1`).

The easiest way to tell that your card is not in the deck is if there are *no* cards in the deck, which is the case if `high` is less than `low`. Well, there are still cards in the deck, of course, but what I mean is that there are no cards in the segment of the deck indicated by `low` and `high`.

With that instruction added, the function works correctly:

```
define find_bisect(deck, card, low, high);
    printf(low >< ', ' >< high, '%p\n');

    if high < low then
        return(-1);
    endif;

    lvars mid = (high + low) div 2;
    lvars comp = compare_card (deck(mid), card);

    if comp = 0 then
        return(mid);
    elseif comp > 0 then
        return(find_bisect (deck, card, low, mid - 1));
    else
        return(find_bisect (deck, card, mid + 1, high));
    endif;
enddefine;
```

I added a print statement at the beginning so I could watch the sequence of recursive calls and convince myself that it would eventually reach the base case. I tried out the following code:

```
lvars card1 = consCard (2, 11);
printf (find_bisect (deck, card1, 1, 52), '%p\n');
```

And got the following output:

```
1, 52
1, 25
14, 25
20, 25
23, 25
24
```

Then I made up a card that is not in the deck (the 15 of Diamonds), and tried to find it. I got the following:

```
1, 52
27, 52
27, 38
27, 31
27, 28
27, 26
-1
```

These tests don't prove that this program is correct. In fact, no amount of testing can prove that a program is correct. On the other hand, by looking at a few cases and examining the code, you might be able to convince yourself.

The number of recursive calls is fairly small, typically 6 or 7. That means we only had to invoke `compareCard` 6 or 7 times, compared to up to 52 times if we did a linear search. In general, bisection is much faster than a linear search, especially for large arrays.

Two common errors in recusive programs are forgetting to include a base case and writing the recursive call so that the base case is never reached. Either error will cause an infinite recursion, in which case Pop11 will (eventually) a stack overflow mishap.

12.9 Decks and subdecks

Looking at the arguments to `find_bisect`

```
find_bisect (deck, card, low, high)
```

it might make sense to treat three of the parameters, `deck`, `low` and `high`, as a single parameter that specifies a **subdeck**.

This kind of thing is quite common, and I sometimes think of it as an **abstract parameter**. What I mean by "abstract," is something that is not literally part of the program text, but which describes the function of the program at a higher level.

For example, when you invoke a function and pass an array and the bounds `low` and `high`, there is nothing that prevents the invoked function from accessing parts of the array that are out of bounds. So you are not literally sending a subset of the deck; you are really sending the whole deck. But as long as the recipient plays by the rules, it makes sense to think of it, abstractly, as a subdeck.

There is one other example of this kind of abstraction that you might have noticed in Section 10.5, when I referred to an "empty" data structure. The reason I put "empty" in quotation marks was to suggest that it is not literally accurate. All variables have values all the time. When you create them, they are given default values. So there is no such thing as an empty object.

But if the program guarantees that the current value of a variable is never read before it is written, then the current value is irrelevant. Abstractly, it makes sense to think of such a variable as "empty."

This kind of thinking, in which a program comes to take on meaning beyond what is literally encoded, is a very important part of thinking like a computer scientist. Sometimes, the word "abstract" gets used so often and in so many contexts that it comes to lose its meaning. Nevertheless, abstraction is a central idea in computer science (as well as many other fields).

A more general definition of "abstraction" is "The process of modeling a complex system with a simplified description in order to suppress unnecessary details while capturing relevant behavior."

12.10 Glossary

encode: To represent one set of values using another set of values, by constructing a mapping between them.

shallow equality: Equality of references. Two references that point to the same object.

deep equality: Equality of values. Two references that point to objects that have the same value.

abstract parameter: A set of parameters that act together as a single parameter.

abstraction: The process of interpreting a program (or anything else) at a higher level than what is literally represented by the code.

Chapter 13

Objects of Arrays

In the previous chapter, we worked with an array of objects, but I also mentioned that it is possible to have an object that contains an array as an instance variable. In this chapter I am going to create a new object, called a `Deck`, that contains an array of `Card`s as an instance variable.

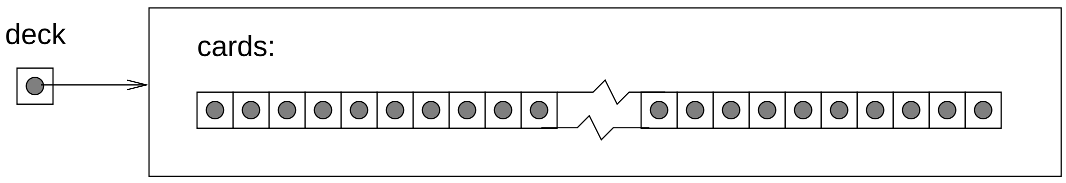
The class definition looks like this

```
define :class Deck;
   slot cards = [];
enddefine;
```

To create a new `Deck` we need extra constructor function

```
define make_Deck(n);
   lvars result = newDeck();
   initv(n) -> cards(result);
   return(result);
enddefine;
```

The name of the instance variable is `cards` to help distinguish the `Deck` object from the array of `Card`s that it contains. Here is a state diagram showing what a `Deck` object looks like with no cards allocated:



As usual, the constructor initializes the instance variable, but in this case it uses the `initv` command to create the array of cards. It doesn't create any cards to go in it, though. For that we could write another constructor that creates a standard 52-card deck and populates it with `Card` objects:

```
define make_full_Deck();
    lvars result = newDeck();
    lvars deck = initv(52);
    deck -> cards(result);

    lvars index = 1;
    lvars suit, rank;
    for suit from 1 to 4 do
        for rank from 1 to 13 do
            consCard(suit, rank) -> deck(index);
            index + 1 -> index;
        endfor;
    endfor;
    return(result);
enddefine;
```

Notice how similar this function is to `build_deck`, except that we had to create empty `Deck` in the first line. Now we invoke it,

```
lvars deck = make_full_Deck ();
```

Now that we have a `Deck` class, it makes sense to change all the functions that pertain to `Deck`s so that they acceps `Deck`s as arguments. Looking at the functions we have written so far, one obvious candidate is `print_deck` (Section 12.7). Here's how it looks, rewritten to work with a `Deck` object:

```
define print_deck(deck);
    lvars i;
    for i from 1 to length(cards(deck)) do
       print_card(cards(deck)(i));
    endfor;
enddefine;
```

The first change is that we can no longer use `length(deck)` to get the length of the array, because `deck` is a `Deck` object now, not an array. It contains an array, but it is not, itself, an array. Therefore, we have to write `length(cards(deck))` to extract the array from the `Deck` object and get the length of the array.

For the same reason, we have to use `cards(deck)(i)` to access an element of the array, rather than just `deck(i)`.

To say the truth, Pop11 gives me enough tools that I could make `Deck` to behave like an array, but I *do not want to do so* – the whole point of defining `Deck` is to distinguish it from plain array!

13.1 Shuffling

For most card games you need to be able to shuffle the deck; that is, put the cards in a random order. In Section 11.6 we saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle, which is usually by dividing the deck in two and then reassembling the deck by choosing alternately from each deck. Since humans usually don't shuffle perfectly, after about 7 iterations the order of the deck is pretty well randomized. But a computer program would have the annoying property of doing a perfect shuffle every time, which is not really very random. In fact, after 8 perfect shuffles, you would find the deck back in the same order you started in. For a discussion of that claim, see `http://www.wiskit.com/marilyn/craig.html` or do a web search with the keywords "perfect shuffle."

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration choose two cards and swap them.

Here is an outline of how this algorithm works. To sketch the program, I am using a combination of Pop11 statements and English words that is sometimes called **pseudocode**:

```
for i from 1 to length(cards(deck)) do
    ;;; choose a random number between i and deck.cards.length
    ;;; swap the ith card and the randomly-chosen card
endfor;
```

The nice thing about using pseudocode is that it often makes it clear what functions you are going to need. In this case, we need something like `random_int`, which chooses a random integer between the parameters `low` and `high`, and `swap_cards` which takes two indices and switches the cards at the indicated positions.

You can probably figure out how to write `random_int` by looking at Section 11.6, although you will have to be careful about possibly generating indices that are out of range.

You can also figure out `swap_cards` yourself. The only tricky thing is to decide whether to swap just the references to the cards or the contents of the cards. Does it matter which one you choose? Which is faster?

I will leave the remaining implementation of these functions as an exercise to the reader.

13.2 Sorting

Now that we have messed up the deck, we need a way to put it back in order. Ironically, there is an algorithm for sorting that is very similar to the algorithm for shuffling. This algorithm is sometimes called **selection sort** because it works by traversing the array repeatedly and selecting the lowest remaining card each time.

During the first iteration we find the lowest card and swap it with the card in the 0th position. During the `ith`, we find the lowest card to the right of `i` and swap it with the `ith` card.

Here is pseudocode for selection sort:

```
    for i from 1 to length(cards(deck)) do
        ;;; find the lowest card at or to the right of i
        ;;; swap the ith card and the lowest card
    endfor;
```

Again, the pseudocode helps with the design of the **helper functions**. In this case we can use `swap_cards` again, so we only need one new one, called `find_lowest_card`, that takes an array of cards and an index where it should start looking.

Once again, I am going to leave the implementation up to the reader.

13.3 Subdecks

How should we represent a hand or some other subset of a full deck? One good choice is to make a `Deck` object that has fewer than 52 cards.

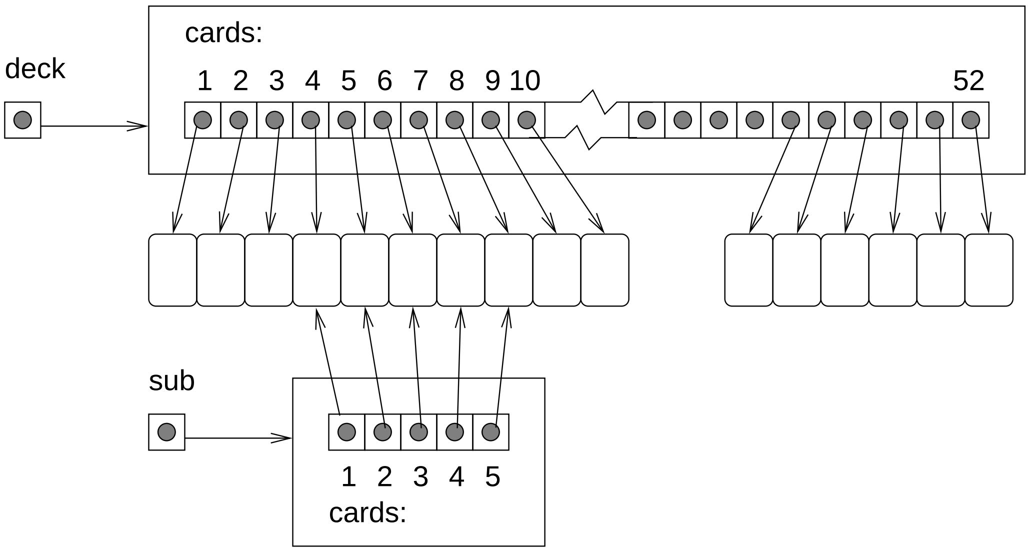
We might want a function, `subdeck`, that takes an array of cards and a range of indices, and that returns a new array of cards that contains the specified subset of the deck:

```
define subdeck(deck, low, high);
    lvars sub = make_Deck(high - low + 1);
    lvars i;
    for i from 1 to length(cards(sub)) do
        cards(deck)(low + i - 1) -> cards(sub)(i);
    endfor;
    return(sub);
enddefine;
```

The length of the subdeck is `high-low+1` because both the low card and high card are included. Similarly, we use `low + i - 1` as an index to `cards(deck)` because `i` starts at `1`. This sort of computation can be confusing, and lead to "off-by-one" errors. Drawing a picture is usually the best way to avoid them.

The `make_Deck` contructor only allocates the array and doesn't allocate any cards. Inside the `for` loop, the subdeck gets populated with copies of the references from the deck.

The following is a state diagram of a subdeck being created with the parameters `low=4` and `high=8`. The result is a hand with 5 cards that are shared with the original deck; i.e. they are aliased.



I have suggested that aliasing is not generally a good idea, since changes in one subdeck will be reflected in others, which is not the behavior you would expect from real cards and decks. But if the objects in question are immutable, then aliasing can be a reasonable choice. In this case, there is probably no reason ever to change the rank or suit of a card. Instead we will create each card once and then treat it as an immutable object. So for `Card`s aliasing is a reasonable choice.

As an exercise, write a version of `find_bisect` that takes a subdeck as an argument, rather than a deck and an index range. Which version is more error-prone? Which version do you think is more efficient?

13.4 Shuffling and dealing

In Section 13.1 I wrote pseudocode for a shuffling algorithm. Assuming that we have a function called `shufflei_deck` that takes a deck as an argument and shuffles it, we can create and shuffle a deck:

```
    lvars deck = make_full_Deck ();
    shuffle_deck (deck);
```

Then, to deal out several hands, we can use `subdeck`:

```
    lvars hand1 = subdeck (deck, 1, 5);
    lvars hand2 = subdeck (deck, 6, 10);
    lvars pack = subdeck (deck, 11, 52);
```

This code puts the first 5 cards in one hand, the next 5 cards in the other, and the rest into the pack.

When you thought about dealing, did you think we should give out one card at a time to each player in the round-robin style that is common in real card games? I thought about it, but then realized that it is unnecessary for a computer program. The round-robin convention is intended to mitigate imperfect shuffling and make it more difficult for the dealer to cheat. Neither of these is an issue for a computer.

This example is a useful reminder of one of the dangers of engineering metaphors: sometimes we impose restrictions on computers that are unnecessary, or expect capabilities that are lacking, because we unthinkingly extend a metaphor past its breaking point. Beware of misleading analogies.

13.5 Mergesort

In Section 13.2, we saw a simple sorting algorithm that turns out not to be very efficient. In order to sort n items, it has to traverse the array n times, and each traversal takes an amount of time that is proportional to n. The total time, therefore, is proportional to n^2.

In this section I will sketch a more efficient algorithm called **mergesort**. To sort n items, mergesort takes time proportional to $n \log n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log n$ can be enormous. Try out a few values of n and see.

The basic idea behind mergesort is this: if you have two subdecks, each of which has been sorted, it is easy (and fast) to merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two subdecks with about 10 cards each and sort them so that when they are face up the lowest cards are on top. Place both decks face up in front of you.

2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.

3. Repeat step two until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. Here's what this looks like in pseudocode:

```
define merge(d1, d2);
    ;;; create a new deck big enough for all the cards
    lvars result = make_Deck (length(cards(d1)) + length(cards(d2)));

    ;;; use the index i to keep track of where we are in
    ;;; the first deck, and the index j for the second deck
    lvars i = 0;
    lvars j = 0;
    ;;; the index k traverses the result deck
    lvars k;
    for k from 0 to length(cards(result)) do

      ;;; if d1 is empty, d2 wins; if d2 is empty, d1 wins;
      ;;; otherwise, compare the two cards

      ;;; add the winner to the new deck
    endfor;
    return(result);
enddefine;
```

The best way to test `merge` is to build and shuffle a deck, use subdeck to form two (small) hands, and then use the sort routine from the previous chapter to sort the two halves. Then you can pass the two halves to `merge` to see if it works.

If you can get that working, try a simple implementation of `merge_sort`:

```
define merge_sort(deck);
   ;;; find the midpoint of the deck
   ;;; divide the deck into two subdecks
   ;;; sort the subdecks using sort_deck
   ;;; merge the two halves and return the result
enddefine;
```

Then, if you get that working, the real fun begins! The magical thing about mergesort is that it is recursive. At the point where you sort the subdecks, why should you invoke the old, slow version of `sort`? Why not invoke the spiffy new `merge_sort` you are in the process of writing?

Not only is that a good idea, it is *necessary* in order to achieve the performance advantage I promised. In order to make it work, though, you have to add a base case so that it doesn't recurse forever. A simple base case is a subdeck with 0 or 1 cards. If `merge_sort` receives such a small subdeck, it can return it unmodified, since it is already sorted.

The recursive version of `merge_sort` should look something like this:

```
define merge_sort (deck);
    ;;; if the deck is 0 or 1 cards, return it
    ;;; find the midpoint of the deck
    ;;; divide the deck into two subdecks
    ;;; sort the subdecks using mergesort
    ;;; merge the two halves and return the result
enddefine;
```

As usual, there are two ways to think about recursive programs: you can think through the entire flow of execution, or you can make the "leap of faith." I have deliberately constructed this example to encourage you to make the leap of faith.

When you were using `sort_deck` to sort the subdecks, you didn't feel compelled to follow the flow of execution, right? You just assumed that the `sort_deck` function would work because you already debugged it. Well, all you did to make `merge_sort` recursive was replace one sort algorithm with another. There is no reason to read the program differently.

Well, actually you have to give some thought to getting the base case right and making sure that you reach it eventually, but other than that, writing the recursive version should be no problem. Good luck!

13.6 Glossary

pseudocode: A way of designing programs by writing rough drafts in a combination of English and Pop11.

helper function: Often a small function that does not do anything enormously useful by itself, but which helps another, more useful, function.

Chapter 14

Object-oriented programming – incomplete chapter

14.1 Programming languages and styles

There are many programming languages in the world, and almost as many programming styles (sometimes called paradigms). Three styles that have appeared in this book are procedural, functional, and object-oriented. Pop11 is a multi-paradigm languages, it is natural to write Pop11 programs in any style. The style I have demonstrated in this book is pretty much procedural. Existing Pop11 programs and the built-in Pop11 packages are written in a mixture of all three styles.

It's not easy to define what object-oriented programming is, but here are some of its characteristics:

- Object definitions (classes) usually correspond to relevant real-world objects. For example, in Chapter 13, the creation of the `Deck` class was a step toward object-oriented programming.
- The majority of functions are generic functions (methods) rather than ordinary functions. Unlike ordinary functions you may have different methods of the same name. When you invoke a method the actual method to use is determined ay runtime depending on classes of arguments. So far all the functions we have written have been ordinary functions. In this chapter we will write some methods.
- The language feature most associated with object-oriented programming is **inheritance**. I will cover inheritance later in this chapter.

Recently object-oriented programming has become quite popular, and there are people who claim that it is superior to other styles in various ways. I hope that by exposing you to a variety of styles I have given you the tools you need to understand and evaluate these claims.

14.2 Ordinary functions and methods

Anything that can be written as a method can also be written as an ordinary function, and vice versa. Sometimes it is just more natural to use one or the other. For reasons that will be clear soon, methods are often shorter than the corresponding ordinary functions.

14.3 The `print_instance` method

There are two methods that are common to many object types: `print_instance` and `=_instance`. `print_instance` converts the object to some reasonable string representation and prints it (`print_instance` is also used to convert object to string). `=_instance` is used to compare objects.

When you print an object using `printf` or print arrow, Pop11 checks to see whether you have provided an object method named `print_instance`, and if so it invokes it. If not, it invokes a default version of `print_instance` that produces the output described in Section 10.4.

Whenever you pass an object to `printf`, Pop11 invokes the `print_instance` method on that object.

You can also convert object to string using `><` operator— Pop11 converts object to string by printing it and collecting characters that would otherwise go to the screen.

```
lvars str = '' >< x;
```

14.4 The `=_instance` method

When you use the `==` operator to compare two objects, what you are really asking is, "Are these two things the same object?" That is, do both objects refer to the same location in memory.

The `=` opertator by default recursively compares components — for many types, this is appropriate definition of equality. For example, two complex numbers are equal if their real parts are equal and their imaginary parts are equal. However, for some classes you need special definition of equality.

When you create a new object type, you can provide your own definition of equality by providing a method called `=_instance`.

The documentation of `=_instance` provides some guidelines you should keep in mind when you make up your own definition of equality:

> The `=_instance` method implements an equivalence relation:

- It is reflexive: for any reference value `x`, `x = x` should return `true`.
- It is symmetric: for any reference values `x` and `y`, `x = y` should return `true` if and only if `y = x` returns `true`.
- It is transitive: for any reference values `x`, `y`, and `z`, if `x = y` returns `true` and `y = z` returns `true`, then `x = z` should return `true`.
- It is consistent: for any reference values `x` and `y`, multiple invocations of `x = y` consistently return `true` or consistently return `false`.

14.5 Invoking one method from another

As you might expect, it is legal and common to invoke one method from another.

14.6 Oddities and errors

14.7 Inheritance

The language feature that is most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of a previously-defined class (including built-in classes).

The primary advantage of this feature is that you can add new methods or instance variables to an existing class without modifying the existing class. This is particularly useful for built-in classes, since sometimes you can't modify them even if you want to.

The reason inheritance is called "inheritance" is that the new class inherits all the instance variables and methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class.

14.8 The class hierarchies

14.9 Object-oriented design

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of build-in classes without having to modify them.

On the other hand, inheritance can make programs difficult to read, since it is sometimes not clear, when a method is invoked, where to find the definition.

Also, many of the things that can be done using inheritance can be done almost as elegantly (or more so) without it.

14.10 Glossary

method: A function that has multiple versions, which version is used depends on argument types (classes).

Chapter 15

Linked lists

15.1 References in objects

In the last chapter we saw that the instance variables of an object can be arrays, and I mentioned that they can be objects, too.

One of the more interesting possibilities is that an object can contain a reference to another object of the same type. There is a common data structure, the **list**, that takes advantage of this feature.

Lists are made up of **nodes**, where each node contains a reference to the next node in the list. In addition, each node usually contains a unit of data called the **cargo**. In our first example, the cargo will be a single integer, but later we will write a **generic** list that can contain objects of any type.

Pop11 has a builtin list type – in many situations builtin lists are more convenient. However, we want to show here object oriented approach to lists. Moreover, the techniques we present generalize to more complicated problems, where builtin Pop11 data structures are not sufficient.

15.2 The `Node` class

As usual when we write a new class, we'll start with the instance variables. We also define `print_instance` method.

```
define :class Node;
    slot cargo = 0;
    slot next = [];
enddefine;

define :method print_instance(x : Node);
    printf(cargo, '%p');
enddefine;
```

The declarations of the instance variables follow naturally from the specification.

To test the implementation so far, we use something like this:

```
lvars node = consNode (1, []);
printf(node, '%p');
```

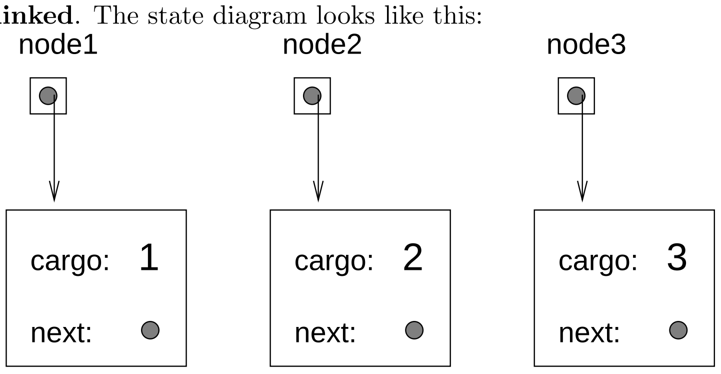
The result is simply

```
1
```

To make it interesting, we need a list with more than one node!

```
lvars node1 = consNode (1, []);
lvars node2 = consNode (2, []);
lvars node3 = consNode (3, []);
```

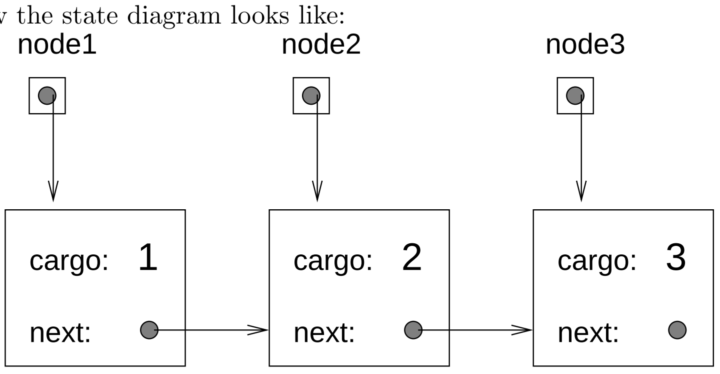
This code creates three nodes, but we don't have a list yet because the nodes are not **linked**. The state diagram looks like this:



To link up the nodes, we have to make the first node refer to the second and the second node refer to the third.

```
node2 -> next(node1);
node3 -> next(node2);
[] -> next(node3);
```

The reference of the third node is `[]`, which indicates that it is the end of the list. Now the state diagram looks like:



Now we know how to create nodes and link them into lists. What might be less clear at this point is why.

15.3 Lists as collections

The thing that makes lists useful is that they are a way of assembling multiple objects into a single entity, sometimes called a collection. In the example, the first node of the list serves as a reference to the entire list.

If we want to pass the list as a parameter, all we have to pass is a reference to the first node. For example, the method `print_list` takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end (indicated by the `[]`).

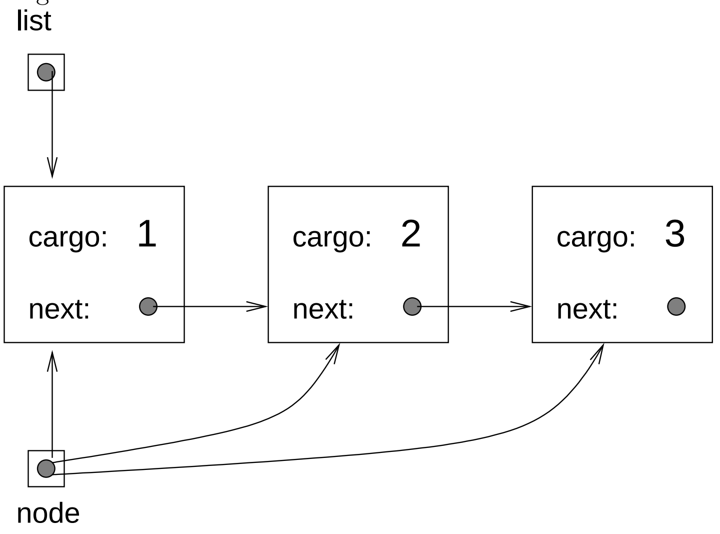
```
define print_list(list);
    lvars node = list;
    while node /= [] do
        printf(node, '%p');
        next(node) -> node;
    endwhile;
    printf('\n');
enddefine;
```

To invoke this method we just have to pass a reference to the first node:

```
print_list(node1);
```

Inside `print_list` we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the `next` value from each node to get to the next node.

This diagram shows the value of `list` and the values that `node` takes on:



This way of moving through a list is called a **traversal**, just like the similar pattern of moving through the elements of an array. It is common to use a loop variable like `node` to refer to each of the nodes in the list in succession.

The output of this function is

```
123
```

Better way to print list is to print them in parentheses with commas between the elements, as in `(1, 2, 3)`. As an exercise, modify `print_list` so that it generates output in this format.

15.4 Lists and recursion

Recursion and lists go together like fava beans and a nice Chianti. For example, here is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head) and the rest (called the tail).
2. Print the tail backwards.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backwards. But *if* we assume that the recursive call works—the leap of faith—then we can convince ourselves that this algorithm works.

All we need is a base case, and a way of proving that for any list we will eventually get to the base case. A natural choice for the base case is a list with a single element, but an even better choice is the empty list, represented by null.

```
define print_backward (list);
    if list == [] then
        return;
    endif;

    lvars head = list;
    lvars tail = next(list);

    print_backward (tail);
    printf(head, '%p');
enddefine;
```

The first line handles the base case by doing nothing. The next two lines split the list into `head` and `tail`. The last two lines print the list.

We invoke this method exactly as we invoked `print_list`:

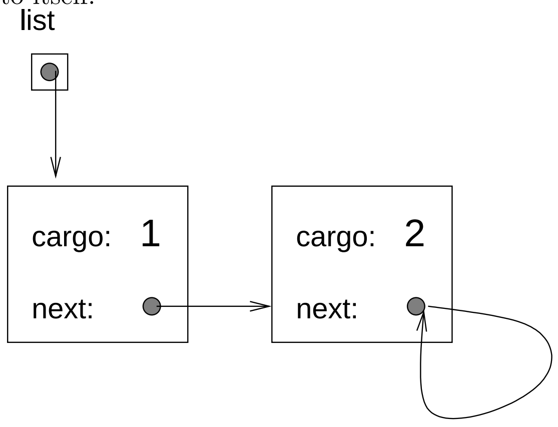
```
print_backward (node1);
```

The result is a backwards list.

Can we prove that this method will always terminate? In other words, will it always reach the base case? In fact, the answer is no. There are some lists that will make this method crash.

15.5 Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself.



If we invoke `print_list` on this list, it will loop forever. If we invoke `print_backward` it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that `print_list` and `print_backward` terminate. The best we can do is the hypothetical statement, "If the list contains no loops, then these methods will terminate." This sort of claim is called a **precondition**. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. We will see more examples soon.

15.6 The fundamental ambiguity theorem

There is a part of `print_backward` that might have raised an eyebrow:

```
    lvars head = list;
    lvars tail = next(list);
```

After the first assignment, `head` and `list` have the same value. So why did I create a new variable?

The reason is that the two variables play different roles. We think of `head` as a reference to a single node, and we think of `list` as a reference to the first node of a list. These "roles" are not part of the program; they are in the mind of the programmer.

The second assignment creates a new reference to the second node in the list, but in this case we think of it as a list. So, head and tail play different roles.

This ambiguity is useful, but it can make programs with lists difficult to read. I often use variable names like node and list to document how I intend to use a variable, and sometimes I create additional variables to disambiguate.

I could have written print_backward without head and tail, but I think it makes it harder to understand:

```
define print_backward (list);
    if list == [] then
        return;
    endif;

    print_backward (next(list));
    printf(list, '%p');
enddefine;
```

Looking at the two function calls, we have to remember that print_backward treats its argument as a list and print treats its argument as a single object.

Always keep in mind the **fundamental ambiguity theorem**:

> A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.

15.7 Object methods for nodes

You might have wondered why print_list and print_backward are not methods. I have made the claim that anything that can be done with functions can also be done with object methods; it's just a question of which form is cleaner.

In this case there is a legitimate reason to choose functions. It is works to send [] as an argument to a function, but [] is not an instance of class Node. So, if we change print_list to a method working on Nodes we need a separate method to handle [].

Before we do that we need to declare extra class:

```
define :mixin EmptyList;
enddefine;
```

Now we say that empty list is a member of this new class:

```
define :extant nil is EmptyList;
enddefine;
```

Finally, we can define print_list:

```
define :method print_list(list : EmptyList);
    printf('()');
enddefine;
```

This limitation makes it awkward to write list-manipulating code in a clean, object-oriented style. A little later we will see a way to get around this, though.

As a less clean alternative we can define `print_list` and `print_backward` as methods which are not specific to any class – just add the keyword `:method` the the definition:

```
define :metod print_backward (list);
    if list == [] then
        return;
    endif;

    print_backward (next(list));
    printf(list, '%p');
enddefine;
```

This definition will be used for all classes that do not have their own `print_backward` method. Of course, if a class has its own `print_backward` it will use its own method.

15.8 Modifying lists

Obviously one way to modify a list is to change the cargo of one on the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes.

As an example, we'll write a method that removes the second node in the list and returns a reference to the removed node.

```
define remove_second(list);
    lvars first = list;
    lvars second = next(node);

    ;;; make the first node refer to the third
    next(second) -> next(first);

    ;;; separate the second node from the rest of the list
    [] -> next(second);
    return(second);
enddefine;
```

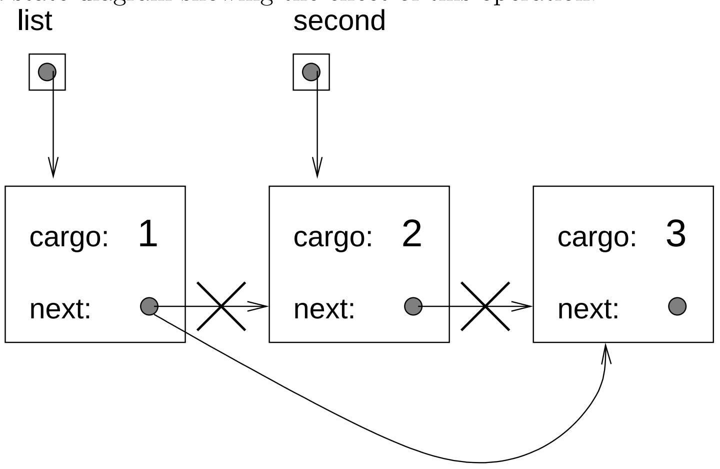
Again, I am using temporary variables to make the code more readable. Here is how to use this method.

```
print_list (node1);
lvars removed = remove_second (node1);
print_list (removed);
print_list (node1);
```

The output is

```
(1, 2, 3)           the original list
(2)                 the removed node
(1, 3)              the modified list
```

Here is a state diagram showing the effect of this operation.



What happens if we invoke this method and pass a list with only one element (a **singleton**)? What happens if we pass the empty list as an argument? Is there a precondition for this method?

15.9 Wrappers and helpers

For some list operations it is useful to divide the labor into two methods. For example, to print a list backwards in the conventional list format, `(3, 2, 1)` we can use the `print_backwards` method to print `3, 2,` but we need a separate method to print the parentheses and the first node. We'll call it `print_backward_nicely`.

```
define print_backward_nicely(list);
    printf('(');
    if list /== [] then
        lvars head = list;
        lvars tail = next(list);
        print_backward (tail);
        printf(head, '%p');
    endif;
    printf(')\n');
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton.

Elsewhere in the program, when we use this method, we will invoke `print_backward_nicely` directly and it will invoke `print_backward` on our behalf. In that sense, `print_backward_nicely` acts as a **wrapper**, and it uses `print_backward` as a helper.

15.10 The `LinkedList` class

There are a number of subtle problems with the way we have been implementing lists. In a reversal of cause and effect, I will propose an alternative implementation first and then explain what problems it solves.

First, we will create a new class called `LinkedList`. Its instance variables are an integer that contains the length of the list and a reference to the first node in the list. LinkedList objects serve as handles for manipulating lists of Node objects.

```
define :class LinkedList;
    slot length = 0;
    slot head = [];
enddefine;
```

One nice thing about the `LinkedList` class is that it gives us a natural place to put wrapper functions like `printBackwardNicely`, which we can make an object method in the `LinkedList` class.

```
define :method print_backward(list : LinkedList);
    printf('(');
    if head(list) /== [] then
        lvars tail = next(head(list));
        print_backward (tail);
        printf(head(list), '%p');
    endif;
    printf(')\n');
enddefine;
```

Just to make things confusing, I renamed `print_backward_nicely`. Now there are two methods named `print_backward`: a "catch all" which handles `Node` class (the helper) and one in the `LinkedList` class (the wrapper).

So, one of the benefits of the `LinkedList` class is that it provides a nice place to put wrapper functions. Another is that it makes it easier to add or remove the first element of a list. For example, `add_first` is a method for `LinkedLists`; it takes an `iitem` as an argument and puts it at the beginning of the list.

```
define :method add_first(list : LinkedList, item)
    lvars node = consNode (item, head(list));
    node -> head(list);
    length(list) + 1 -> length(list);
enddefine;
```

As always, to check code like this it is a good idea to think about the special cases. For example, what happens if the list is initially empty?

As an excercise write methods named `empty`, `add_last` and `remove_first` which respectively test if a `LinkedList` contains no elements, add an element at the end of list and removes first element of the list.

15.11 Invariants

Some lists are "well-formed;" others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the `length` value in the `LinkedList` object should be equal to the actual number of nodes in the list.

Requirements like this are called **invariants** because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are some times when they are violated. For example, in the middle of `add_first`, after we have added the node, but before we have incremented `length`, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normally the requirement is that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

15.12 Glossary

list: A data structure that implements a collection using a sequence of linked nodes.

node: An element of a list, usually implemented as an object that contains a reference to another object of the same type.

cargo: An item of data contained in a node.

link: An object reference embedded in an object.

generic data structure: A kind of data structure that can contain data of any type.

precondition: An assertion that must be true in order for a method to work correctly.

invariant: An assertion that should be true of an object at all times (except maybe while the object is being modified).

wrapper method: A method that acts as a middle-man between a caller and a helper method, often offering an interface that is cleaner than the helper method's.

Chapter 16

Stacks

16.1 Abstract data types

The data types we have looked at so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the `Card` class represents a card using two integers. As I discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do) but it does not not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.

- Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.

- Well-known ADTs, like the `Stack` ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.

- The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the **client** code, from the code that implements the ADT, called **provider** code because it provides a standard set of services.

16.2 The Stack ADT

In this chapter we will look at one common ADT, the stack. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include arrays and lists.

As I said, an ADT is defined by the operations you can perform on it. Stacks can perform only the following operations:

constructor: Create a new, empty stack.

`push`: Add a new item to the stack.

`pop`: Remove and return an item from the stack. The item that is returned is always the last one that was added.

`empty`: Check whether the stack is empty.

A stack is sometimes called a "last in, first out," or LIFO data structure, because the last item added is the first to be removed.

16.3 Using the Stack object

We show the implementation of the stack later. Here we assume that the definitions from the implementation part are loaded (you can propend them at the beggining of your program). Then the syntax for constructing a new `Stack` is

```
lvars stack = make_Stack();
```

Initially the stack is empty, as we can confirm with the `empty` method, which returns a `boolean`:

```
printf(empty(stack), '%p\n');
```

A stack is a generic data structure, which means that we can add any type of item to it.

First, we push an integer

```
push(stack, 666);
```

We can remove an element from the stack with the `pop` method

```
pop(stack) =>
```

We see that printed return value `666` is the same as the value we `push`ed.

For our next example, we'll will push all items from a list. Let's start by creating and printing a short list.

```
lvars list = [1 2 3];
list =>
```

The output is `[1, 2, 3]`.

The following loop traverses the list and pushes all the items from the list onto the stack:

```
lvars val;
for val in list do
    push(stack, val);
endfor;
```

The following loop is a common idiom for popping all the elements from a stack, stopping when it is empty:

```
while not(empty(stack)) do
   lvars val = pop(stack);
   printf(val, '%p ');
endwhile;
```

The output is `3 2 1`. In other words, we just used a stack to print the elements of a list backwards! Granted, it's not the standard format for printing a list, but using a stack it was remarkably easy to do.

You should compare this code to the implementations of `print_backward` in the previous chapter. There is a natural parallel between the recursive version of `print_backward` and the stack algorithm here. The difference is that `print_backward` uses the run-time stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, just using a `Stack` object instead of the run-time stack.

16.4 Postfix expressions

In most programming languages, mathematical expressions are written with the operator between the two operands, as in `1+2`. This format is called **infix**. An alternate format used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in `1 2+`.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack.

- Starting at the beginning of the expression, get one term (operator or operand) at a time.
  - If the term is an operand, push it on the stack.
  - If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.
- When we get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

As an exercise, apply this algorithm to the expression `1 2 + 3 *`.

This example demonstrates one of the advantages of postfix: there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write `(1 + 2) * 3`. As an exercise, write a postfix expression that is equivalent to `1 + 2 * 3`?

16.5 Parsing

In order to implement the algorithm from the previous section, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**, and the results—the individual chunks of the string—are called **tokens**.

Pop11 has built-in tokenizer that parses strings and breaks them into tokens.

```
lvars char_rep = stringin('Here are five tokens.');
lvars item_rep = incharitem(char_rep);
```

The following loop is a standard idiom for extracting the tokens from a item repeater.

```
lvars it;
while (itemrep() ->> it) /= termin do
    printf (it, '%p\n');
endwhile;
```

The output is

```
Here
are
four
tokens
.
```

Builtin itemiser works as we want for our expressions.

```
lvars char_rep = stringin('11 22+33*');
```

Now the output is

```
11
22
+
33
*
```

This is just the stream of tokens we would like for evaluating this expression.

16.6 Implementing ADTs

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct—in accord with the specification of the ADT—and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn't worry about the details. When you are using library classes, you have the luxury of thinking exclusively as a client.

When you implement an ADT, on the other hand, you also have to write client code to test it. In that case, you sometimes have to think carefully about which role you are playing at a given instant.

In the next few sections we will switch gears and look at one way of implementing the Stack ADT, using an array. Start thinking like a provider.

16.7 Array implementation of the Stack ADT

The instance variables for this implementation are a vector, which will contain the items on the stack, and an integer index which will keep track of the next available space in the array. Initially, the array is empty and the index is `1`.

To add an element to the stack (`push`), we'll copy a reference to it onto the stack and increment the index. To remove an element (`pop`) we have to decrement the index first and then copy the element out.

Here is the class definition:

```
define :class Stack;
    slot buffer;
    slot index;
enddefine;

define make_Stack();
    return(consStack(initv(128), 1));
enddefine;
```

As usual, once we have chosen the instance variables, it is a mechanical process to write a constructor. For now, the default size is 128 items. Later we will consider better ways of handling this.

Checking for an empty stack is trivial.

```
define :method empty(s : Stack);
   return (index(s) = 1);
enddefine;
```

It it important to remember, though, that the number of elements in the stack is not the same as the size of the array. Initially the size is 128, but the number of elements is 0.

The implementations of push and pop follow naturally from the specification.

```
define :method push(s : Stack, item);
    item ->buffer(s)(index(s));
    index(s) + 1 -> index(s);
enddefine;

define :method pop(s : Stack);
    index(s) - 1 -> index(s);
    return(buffer(s)(index(s)));
enddefine;
```

To test these methods, we the client code from the first part of this chapter.

If everything goes according to plan, the program should work without any additional changes. Again, one of the strengths of using an ADT is that you can change implementations without changing client code.

16.8 Resizing arrays

A weakness of this implementation is that it chooses an arbitrary size for the array when the `Stack` is created. If the user pushes more than 128 items onto the stack, it will cause a `BAD SUBSCRIPT FOR INDEXED ACCESS` mishap.

An alternative is to let the client code specify the size of the array. This alleviates the problem, but it requires the client to know ahead of time how many items are needed, and that is not always possible.

A better solution is to check whether the array is full and make it bigger when necessary. Since we have no idea how big the array needs to be, it is a reasonable strategy to start with a small size and double it each time it overflows.

Here's the improved version of `push`:

```
define :method push(s : Stack, item);
    if full(s) then
        resize(s);
    endif;
    ;;; at this point we can prove that index <= length(buffer)
    item ->buffer(s)(index(s));
    index(s) + 1 -> index(s);
enddefine;
```

Before putting the new item in the array, we check if the array is full. If so, we invoke `resize`. After the `if` statement, we know that either (1) there was room in the array, or (2) the array has been resized and there is room. If `full` and `resize` are correct, then we can prove that `index <= array.length`, and therefore the next statement cannot cause an exception.

Now all we have to do is implement `full` and `resize`.

```
define full(s);
    return (index(s) = length(buffer(s) + 1));
enddefine;

define resize(s);
    lvars old_buffer = buffer(s);
    lvars buffer_length = length(old_buffer);
    lvars new_buffer = initv(buffer_length * 2);

    ;;; we assume that the old_buffer is full
    for i from 1 to buffer_length do
        old_buffer(i) -> new_buffer(i);
    endfor;
    new_buffer -> buffer(s);
enddefine;
```

We declared `full` and `resize` not as methods but as ordinary functions. This is acceptable, because since there is no reason for client code to use these functions, so we can hide them in a separate section.

The implementation of `full` is trivial; it just checks whether the index has gone beyond the range of valid indices.

The implementation of `resize` is straightforward, with the caveat that it assumes that the old array is full. In other words, that assumption is a precondition of this method. It is easy to see that this precondition is satisfied, since the only way `resize` is invoked is if `full` returns true, which can only happen if `index = length(buffer)+1`.

At the end of `resize`, we replace the old array with the new one (causing the old to be garbage collected). The `length` of new `buffer` is twice as big as the old, and `index` hasn't changed, so now it must be true that `index <= length(buffer)`. This assertion is a **postcondition** of `resize`: something that must be true when the method is complete (as long as its preconditions were satisfied).

Preconditions, postconditions, and invariants are useful tools for analyzing programs and demonstrating their correctness. In this example I have demonstrated a programming style that facilitates program analysis and a style of documentation that helps demonstrate correctness.

16.9 Glossary

abstract data type (ADT): A data type (usually a collection of objects) that is defined by a set of operations, but that can be implemented in a variety of ways.

client: A program that uses an ADT (or the person who wrote the program).

provider: The code that implements an ADT (or the person who wrote it).

infix: A way of writing mathematical expressions with the operators between the operands.

postfix: A way of writing mathematical expressions with the operators after the operands.

parse: To read a string of characters or tokens and analyze their grammatical structure.

token: A set of characters that are treated as a unit for purposes of parsing, like the words in a natural language.

delimiter: A character that is used to separate tokens, like the punctuation in a natural language.

predicate: A mathematical statement that is either true or false.

postcondition: A predicate that must be true at the end of a method (provided that the preconditions were true at the beginning).

Chapter 17

Queues and Priority Queues

This chapter presents two ADTs: Queues and Priority Queues. In real life a **queue** is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. For example, at airports customers whose flight is leaving imminently are sometimes taken from the middle of the queue. Also, at supermarkets a polite customer might let someone with only a few items go first.

The rule that determines who goes next is called a **queueing discipline**. The simplest queueing discipline is called **FIFO**, for "first-in-first-out." The more general queueing discipline is **priority queueing**, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is more general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are "fair," but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: a Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queueing policy.

As with most ADTs, there are a number of ways to implement queues. Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections: arrays, lists, or vectors. Our choice among them will be based in part on their performance— how long it takes to perform the operations we want to perform— and partly on ease of implementation.

17.1 The queue ADT

The queue ADT is defined by the following operations:

constructor: Create a new, empty queue.

`insert`: Add a new item to the queue.

`remove:` Remove and return an item from the queue. The item that is returned is the first one that was added.

`empty:` Check whether the queue is empty.

One possible queue implementation, take advantage of the Pop11 lists.

As far as our implementation goes, it does not matter what kind of object is in the `Queue`, so we can make it generic. Here is what the implementation looks like.

```
define :class Queue;
    slot list = [];
enddefine;

define :method empty (q : Queue);
    return list(q) = [];
enddefine;

define :method insert(q : Queue, item);
    list(q) <> [^item] -> list(q);
enddefine;

define :method remove(q : Queue);
    lvars l = list(q);
    back(l) -> list(q);
    return(front(l));
enddefine;
```

A queue object contains a single instance variable, which is the list that implements it.

Similar, slightly simpler implementation uses `LinkedList` class:

```
define :class Queue;
    slot list = newLinkedList();
enddefine;

define :method empty (q : Queue);
    return(empty(list(q)));
enddefine;

define :method insert(q : Queue, item);
    add_last(list(q), item);
enddefine;

define :method remove(q : Queue);
    return(remove_first(list(q)));
enddefine;
```

For each of the methods, all we have to do is invoke one of the methods from the `LinkedList` class.

17.2 Veneer

An implementation like this is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

This example demonstrates one of the nice things about a veneer, which is that it is easy to implement, and one of the dangers of using a veneer, which is the **performance hazard**!

Normally when we invoke a method we are not concerned with the details of its implementation. But there is one "detail" we might want to know—the performance characteristics of the method. How long does it take, as a function of the number of items in the list?

First let's look at `remove_first`.

```
define remove_first(list);
    lvars result = head(list);
    if head(list) /== [] then
        next(head(list)) -> head(list);
        length(list) - 1 -> length(list);
    endif;
    return(result);
enddefine;
```

There are no loops or nontrival function calls here, so that suggests that the run time of this method is the same every time. Such a method is called a **constant time** operation. In reality, the method might be slightly faster when the list is empty, since it skips the body of the conditional, but that difference is not significant.

The performance of `add_last` is very different.

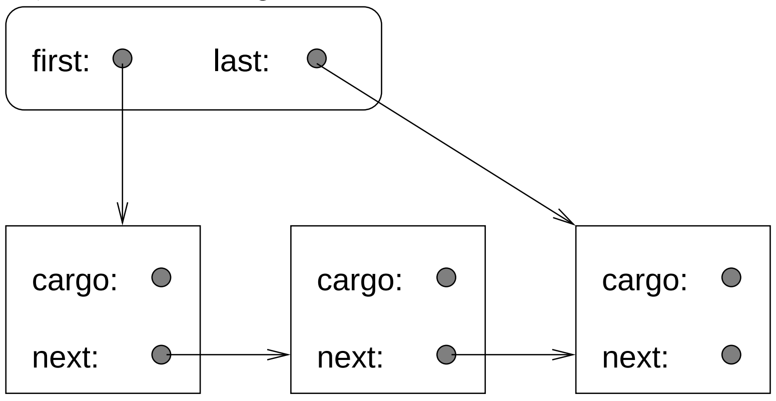
```
define :method add_last(list : LinkedList, item);
    if head(list) == [] then
        consNode(item, []) -> head(list);
        length(list) + 1 -> length(list);
        return;
    endif;
    lvars last = head(list);
    ;;; traverse the list to find the last node
    while next(last) /== [] do
        next(last) -> last;
    endwhile;
    consNode(item, []) -> next(last);
    length(list) + 1 -> length(list);
enddefine;
```

The first conditional handles the special case of adding a new node to an empty list. In this case, again, the run time does not depend on the length of the list. In the general case, though, we have to traverse the list to find the last element so we can make it refer to the new node.

This traversal takes time proportional to the length of the list. Since the run time is a linear function of the length, we would say that this method is **linear time**. Compared to constant time, that's very bad.

17.3 Linked Queue

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to accomplish that is to implement a **linked queue**, which is similar to a linked list in the sense that it is made up of zero or more linked nodes. For simplicity, we will re-use (nodes of) builtin Pop11 lists. The difference is that the queue maintains a reference to both the first and the last node, as shown in the figure.



Here's what a linked `Queue` implementation looks like:

```
define :class Queue;
   slot first = [];
   slot last = [];
enddefine;

define :method empty(q : Queue);
    return(first(q) = []);
enddefine;
```

So far it is straightforward. In an empty queue, both `first` and `last` are null. To check whether a list is empty, we only have to check one of them.

insert is a little more complicated because we have to deal with several special cases.

```
define :method insert(q : Queue, item);
    lvars last_node = last(q);
    lvars node = cons(item, []);
    if last_node /= [] then
        node -> back(last_node);
    endif;
    node -> last(q);
    if first(q) = [] then
        node -> first(q);
    endif;
enddefine;
```

The first condition checks to make sure that last refers to a node; if it does then we have to make it refer to the new node.

The second condition deals with the special case where the list was initially empty. In this case both first and last refer to the new node.

remove also deals with several special cases.

```
define :method remove(q : Queue);
    lvars result = first(q);
    if result /= [] then
        back(result) -> first(q);
    endif;
    if first(q) = [] then
       [] -> last(q);
    endif;
    return(result);
enddefine;
```

The first condition checks whether there were any nodes in the queue. If so, we have to copy the next node into first. The second condition deals with the special case that the list is now empty, in which case we have to make last null.

As an exercise, draw diagrams showing both operations in both the normal case and in the special cases, and convince yourself that they are correct.

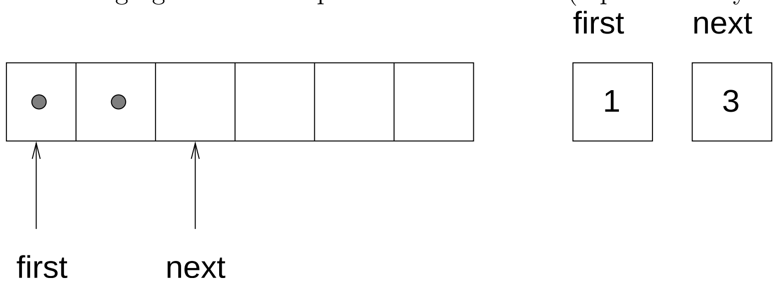
Clearly, this implementation is more complicated than the veneer implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal: both insert and remove are constant time.

17.4 Circular buffer

Another common implementation of a queue is a **circular buffer**. “Buffer” is a general name for a temporary storage location, although it often refers to an array, as it does in this case. What it means to say a buffer is "circular" should become clear in a minute.

The implementation of a circular buffer is similar to the array implementation of a stack, as in Section 16.7. The queue items are stored in an array, and we use indices to keep track of where we are in the array. In the stack implementation, there was a single index that pointed to the next available space. In the queue implementation, there are two indices: `first` points to the space in the array that contains the first customer in line and `next` points to the next available space.

The following figure shows a queue with two items (represented by dots).



There are two ways to think of the variables `first` and `last`. Literally, they are integers, and their values are shown in boxes on the right. Abstractly, though, they are indices of the array, and so they are often drawn as arrows pointing to locations in the array. The arrow representation is convenient, but you should remember that the indices are not references; they are just integers.

Here is an incomplete array implementation of a queue:

```
define :class Queue;
    slot buffer = initv(128);
    slot first = 1;
    slot next = 1;
enddefine;

define :method empty(q : Queue);
    return(first(q) = next(q));
enddefine;
```

The instance variables are straightforward, and the `newQueue` constructor is adequate, although again we have the problem that we have to choose an arbitrary size for the array. Later we will solve that problem, as we did with the stack, by resizing the array if it gets full.

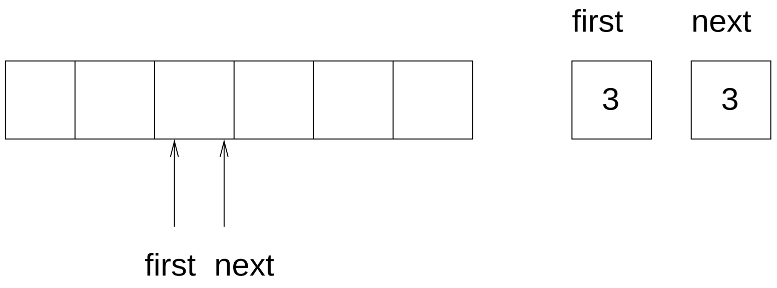
The implementation of `empty` is a little surprising. You might have thought that `first = 1` would indicate an empty queue, but that neglects the fact that the head of the queue is not necessarily at the beginning of the array. Instead, we know that the queue is empty if `head` equals `next`, in which case there are no items left. Once we see the implementation of `insert` and `remove`, that situation will more more sense.

```
define :method insert(q : Queue, item);
    item -> buffer(q)(next(q));
    next(q) + 1 -> next(q);
enddefine;

define :method remove(q : Queue);
    lvars result = buffer(q)(first(q));
    first(q) + 1 -> first(q);
    return(result);
enddefine;
```

`insert` looks very much like `push` in Section 16.7; it puts the new item in the next available space and then increments the index.

`remove` is similar. It takes the first item from the queue and then increments `first` so it refers to the new head of the queue. The following figure shows what the queue looks like after both items have been removed.

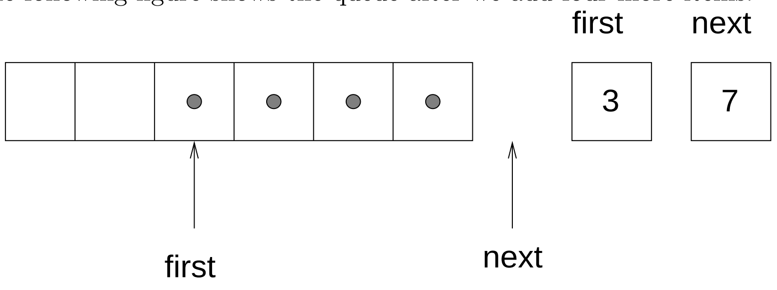


It is always true that `next` points to an available space. If `first` catches up with `next` and points to the same space, then `first` is referring to an "empty" location, and the queue is empty. I put "empty" in quotation marks because it is possible that the location that `first` points to actually contains a value (we do nothing to ensure that empty locations contain `[]`); on the other hand, since we know the queue is empty, we will never read this location, so we can think of it, abstractly, as empty.

As an exercise, fix `remove` so that it returns `[]` if the queue is empty.

The next problem with this implementation is that eventually it will run out of space. When we add an item we increment `next` and when we remove an item we increment `first`, but we never decrement either. What happens when we get to the end of the array?

The following figure shows the queue after we add four more items:



The array is now full. There is no "next available space," so there is nowhere for `next` to point. One possibility is that we could resize the array, as we did with the stack implementation. But in that case the array would keep getting bigger regardless of how many items were actually in queue. A better solution is to wrap around to the beginning of the array and reuse the spaces there. This "wrap around" is the reason this implementation is called a circular buffer.

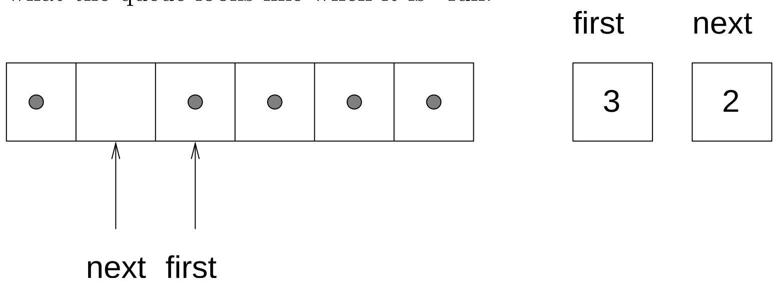
One way to wrap the index around is to add a special case whenever we increment an index:

```
next(q) + 1 -> next(q);
if next(q) > length(buffer(q)) then
    1 -> next(q);
```

A fancy alternative is to use the modulus operator:

```
(next(q) rem length(buffer(q))) + 1 -> next(q);
```

Either way, we have one last problem to solve. How do we know if the queue is *really* full, meaning that we cannot insert another item? The following figure shows what the queue looks like when it is "full."



There is still one empty space in the array, but the queue is full because if we insert another item, then we have to increment `next` such that `next == first`, and in that case it would appear that the queue was empty!

To avoid that, we sacrifice one space in the array. So how can we tell if the queue is full?

```
if (next(q) rem length(buffer(q))) + 1 = first(q) then
```

And what should we do if the array is full? In that case resizing the array is probably the only option.

As an exercise, put together all the code from this section and write an implementation of a queue using a circular buffer that resizes itself when necessary.

17.5 Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. The interface is:

constructor: Create a new, empty queue.

`insert`**:** Add a new item to the queue.

`remove`**:** Remove and return an item from the queue. The item that is returned is the one with the highest priority.

`empty`**:** Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is whatever item in the queue has the highest priority. What the priorities are, and how they compare to each other, are not specified by the Priority Queue implementation. It depends on what the items are that are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might choose from highest to lowest, but if they are golf scores, we would go from lowest to highest.

So we face a new problem. We would like an implementation of Priority Queue that is generic—it should work with any kind of object—but at the same time the code that implements Priority Queue needs to have the ability to compare the objects it contains.

The solution is that the order is defined by a method. So, all object that we store in Priority Queue should have two argument `compare` method.

17.6 Array implementation of Priority Queue

We start with the class definition

```
define :class Priority_Queue;
    slot buffer = initv(16);
    slot index = 1;
enddefine;
```

As usual, `index` is the index of the next available location in the array. The automatilally generated `newPriority_Queue` constructor is again adequate. I chose the initial size for the array arbitrarily.

`empty` is similar to what we have seen before.

```
define :method empty(pq : Priority_Queue);
    return(index(pq) = 1);
enddefine;
```

`insert` is similar to `push`:

```
define :method insert(pq : Priority_Queue, item);
    if index(pq) = length(buffer(pq)) + 1 then
        resize(pq);
    endif;
    item ->buffer(pq)(index(pq));
    index(pq) + 1 -> index(pq);
enddefine;
```

I omitted the implementation of `resize`. The only substantial method in the class is `remove`, which has to traverse the array to find and remove the largest item:

```
define :method remove(pq : Priority_Queue);
    if index(pq) = 1 then
        return([]);
    endif;
    lvars buf = buffer(pq);
    lvars max_index = 1;
    lvars i;

    for i from 2 to index(pq) - 1 do
        if compare(buf(i), buf(max_index)) > 0 then
            i -> max_index;
        endif;
    endfor;
    lvars result = buf(max_index);

    ;;; move the last item into the empty slot
    index(pq) - 1 -> index(pq);
    buf(index(pq)) -> buf(max_index);
    return(result);
enddefine;
```

As we traverse the array, `max_index` keeps track of the index of the largest element we have seen so far. What it means to be the "largest" is determined by `compare`.

17.7 A Priority Queue client

The implementation of Priority Queue is written entirely in terms of `compare` function, but `compare` remains undefined. It is easy to write a special purpose `compare` function, but we would like to use Priority Queue with arbitrary classes, and clearly each such class may need different `compare`. Then solution is to make `compare` a generic function. However, we would like also to store integers in Priority Queue, so we need a compare method for integers. Pop11 has a special mechanizm which turns normal data type into a class.

First, we need an auxilary class

```
define :mixin Integer;
enddefine;
```

Now we say that integer are member of this new class

```
define :extant integer is Integer;
enddefine;

define :extant biginteger is Integer;
enddefine;
```

Now we can define `compare` method

```
define :method compare(x : Integer, y : Integer);
    if x > y then
        return(1);
    elseif x = y then
        return(0);
    else
        return(-1);
    endif;
enddefine;
```

Now, we can start using our Priority Queue.

```
lvars pq = make_Priority_Queue ();
insert(pq, 17);
insert(pq, 12);
insert(pq, 44);
```

This code creates a new, empty Priority Queue. Then it inserts three integers into the queue.

To get items out of the queue, we have to reverse the process:

```
while not(empty(pq)) do
   lvars item = remove(pq);
   printf (item, '%p\n');
endwhile;
```

This loop removes all the items from the queue and prints them.

17.8 The `Golfer` class

Finally, let's looks at how we can make a new class that has `compare` method. As an example of something with an unusual definition of "highest" priority, we'll use golfers:

```
define :class Golfer;
    slot name;
    slot score;
enddefine;
```

The class definition and the constructor are pretty much the same as always.

```
define :method compare(x : Golfer, y : Golfer);
    lvars a = score(x);
    lvars b = score(y);

    ;;; for golfers, low is good!
    if a < b then
        return(1);
    elseif a > b then
        return(-1);
    else
        return(0);
    endif;
enddefine;
```

Finally, we can create some golfers:

```
lvars tiger = consGolfer ('Tiger Woods', 61);
lvars phil = consGolfer ('Phil Mickelson', 72);
lvars hal = consGolfer ('Hal Sutton', 69);
```

And put them in the queue:

```
insert(pq, tiger);
insert(pq, phil);
insert(pq, hal);
```

When we pull them out:

```
while not(empty(pq)) do
    lvars golfer = remove(pq);
    printf(golfer, '%p\n');
endwhile;
```

They appear in descending order (for golfers):

```
<Golfer name:Tiger Woods score:61>
<Golfer name:Hal Sutton score:69>
<Golfer name:Phil Mickelson score:72>
```

When we switched from `Integers` to `Golfers`, we didn't have to make any changes in `Priority_Queue` at all. So we succeeded in maintaining a barrier between `Priority_Queue` and the classes that use it, allowing us to reuse the code without modification. Furthermore, we were able to give the client code control over the definition of `compare`, making this implementation of `Priority_Queue` more versatile.

17.9 The `Comparable` class

Here we discuss one unsatisfactory aspect in our Prority Queue. Namely, we need `compare` method for items that we insert into Prority Queue. However, we can insert any item into Prority Queue. We get runtime error only we we try to remove item from the Prority Queue and the `compare` method is undefined. We would like to get error message where the real error is, that is when we inserted wrong item.

First, we define a new class

```
define :mixin Comparable;
enddefine;
```

The keyword `mixin` means that `Comparable` is **abstract class**—other classes can inherit from `Comparable`, but we are not allowed to create object of `Comparable` class.

Next, we change first line of the `insert` method to read

```
define :method insert(pq : Priority_Queue, item : Comparable);
```

Now, the second argument to `insert` must be `Comparable`

We need to change definition of `Integer` class

```
define :mixin Integer is Comparable;
enddefine;
```

Finally, we need to change first line of definition of `Golfer` class to

```
define :class Golfer is Comparable;
```

Now, let us try to `insert` something that is not `Comparable` like a string

```
insert(pq, 'string is not Comparable');
```

We get error message like `Method "insert" failed`.

However, this is only partial solution, we can still insert integers and `Golfer`s into the same Prority Queue, but we can not `compare` an integer with a `Golfer`. Note: when it make sense we can easily extend compare, but comparing integers and `Golfer`s makes no sense.

17.10 Glossary

queue: An ordered set of objects waiting for a service of some kind.

queueing discipline: The rules that determine which member of a queue is removed next.

FIFO: “first in, first out,” a queueing discipline in which the first member to arrive is the first to be removed.

priority queue: A queueing discipline in which each member has a priority determined by external factors. The member with the highest priority is the first to be removed.

Priority Queue: An ADT that defines the operations one might perform on a priority queue.

veneer: A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

performance hazard: A danger associated with a veneer that some of the methods might be implemented inefficiently in a way that is not apparent to the client.

constant time: An operation whose run time does not depend on the size of the data structure.

linear time: An operation whose run time is a linear function of the size of the data structure.

linked queue: An implementation of a queue using a linked list and references to the first and last nodes.

circular buffer: An implementation of a queue using an array and indices of the first element and the next available space.

abstract class: A set of classes. The abstract class specification lists the requirements a class must satisfy to be included in the set.

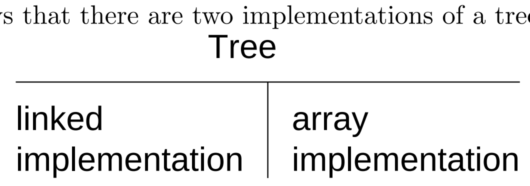
Chapter 18

Trees

This chapter presents a new data structure called a tree, some of its uses and two ways to implement it.

A possible source of confusion is the distinction between an ADT, a data structure, and an implementation of an ADT or data structure. There is no universal answer, because something that is an ADT at one level might in turn be the implementation of another ADT.

To help keep some of this straight, it is sometimes useful to draw a diagram showing the relationship between an ADT and its possible implementations. This figure shows that there are two implementations of a tree:



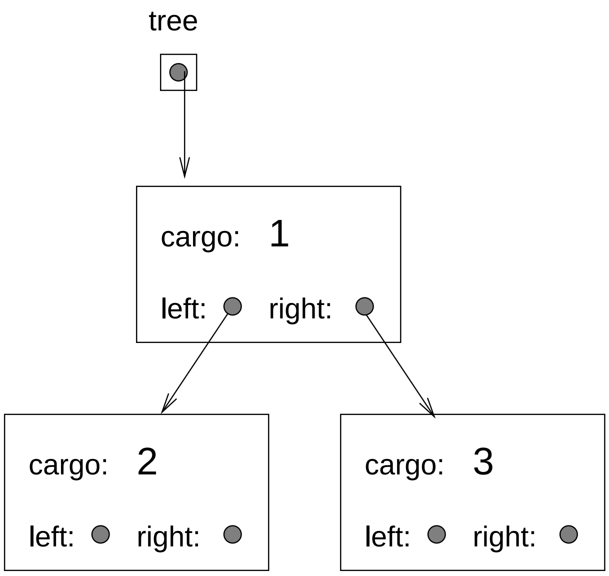
The horizontal line in the figure represents the barrier of abstraction between the ADT and its implementations.

18.1 A tree node

Like lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly null). The class definition looks like this:

```
define :class Tree;
    slot cargo = [];
    slot left = [];
    slot right = [];
enddefine;
```

Like list nodes, tree nodes contain cargo. The other instance variables are named `left` and `right`, in accordance with a standard way to represent trees graphically:



The top of the tree (the node referred to by `tree`) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in yet another metaphor: the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**, and so on.

Finally, there is also a geometric vocabulary for taking about trees. I already mentioned left and right, but there is also "up" (toward the parent/root) and down (toward the children/leaves). Also, all the nodes that are the same distance from the root comprise a **level** of the tree.

I don't know why we need three metaphors for talking about trees, but there it is.

18.2 Building trees

The process of assembling tree nodes is similar to the process of assembling lists. We have automatically defined `consTree` constructor for tree nodes that initializes the instance

We allocate the child nodes first:

```
lvars left_node = consTree(2, [], []);
lvars right_node = consTree(3, [], []);
```

We can create the parent node and link it to the children at the same time:

```
lvars tree = consTree(1, left_node, right_node);
```

This code produces the state shown in the previous figure.

18.3 Traversing trees

By now, any time you see a new data structure, your first question should be, "How can I traverse it?" The most natural way to traverse a tree is recursively. For example, to add up all the integers in a tree, we could write this function:

```
define total(tree);
    if tree = [] then
        return(0);
    else
        return (cargo(tree) + total(left(tree)) + total(right(tree)));
    endif;
enddefine;
```

This is not method because we would like to use `[]` to represent the empty tree, and make the empty tree the base case of the recursion. If the tree is empty, the function returns `0`. Otherwise it makes two recursive calls to find the total value of its two children. Finally, it adds in its own cargo and returns the total.

Although this function works, there is some difficulty fitting it into an object-oriented design. Namely, it should not be part of `Tree` implementation, because it requires the cargo to be numeric.

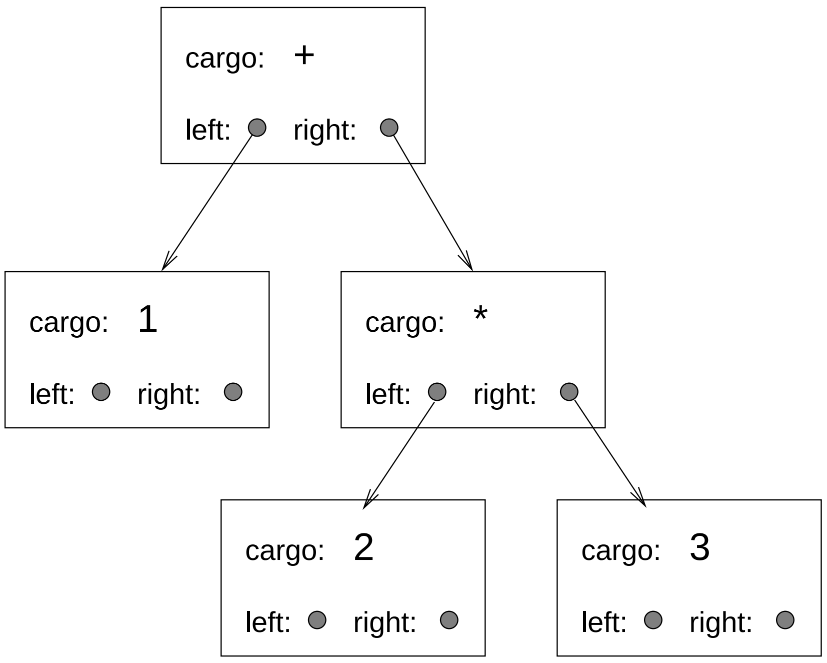
On the other hand, this code accesses the instance variables of the `Tree` nodes, so it "knows" more than it should about the implementation of the tree. If we changed that implementation later (and we will) this code would break.

Later in this chapter we will develop ways to solve this problem, allowing client code to traverse trees containing any kinds of objects without breaking the abstraction barrier between the client code and the implementation. Before we get there, let's look at an application of trees.

18.4 Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the comptation unambiguously. For example, the infix expression `1 + 2 * 3` is ambiguous unless we know that the multiplication happens before the addition.

The following figure represents the same computation:



The nodes can be operands like `1` and `2` or operators like `+` and `*`. Operands are leaf nodes; operator nodes contain references to their operands (all of these operators are **binary**, meaning they have exactly two operands).

Looking at this figure, there is no question what the order of operations is: the multiplication happens first in order to compute the first operand of the addition.

Expression trees like this have many uses. The example we are going to look at is translation from one format (postfix) to another (infix). Similar trees are used inside compilers to parse, optimize and translate programs.

18.5 Traversal

I already pointed out that recursion provides a natural way to traverse a tree. We can print the contents of an expression tree like this:

```
define print(tree);
    if tree = [] then
        return;
    endif;
    printf(cargo(tree), '%p ');
    print(left(tree));
    print(right(tree));
enddefine;
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear before the contents of the children.

For the example expression the output is `+ 1 * 2 3`. This is different from both postfix and infix; it is a new notation called **prefix**, in which the operators appear before their operands.

You might suspect that if we traverse the tree in a different order we get the expression in a different notation. For example, if we print the subtrees first, and then the root node:

```
define print_postorder(tree);
    if tree = [] then
        return;
    endif;
    print_postorder(left(tree));
    print_postorder(right(tree));
    printf(cargo(tree), '%p ');
enddefine;
```

We get the expression in postfix (`1 2 3 * +`)! As the name of the previous method implies, this order of traversal is called **postorder**. Finally, to traverse a tree **inorder**, we print the left tree, then the root, then the right tree:

```
define print_inorder(tree);
    if tree = [] then
        return;
    endif;
    print_inorder(left(tree));
    printf(cargo(tree), '%p ');
    print_inorder(right(tree));
enddefine;
```

The result is `1 + 2 * 3`, which is the expression in infix.

To be fair, I have to point out that I have omitted an important complication. Sometimes when we write an expression in infix we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

18.6 Encapsulation

As I mentioned before, there is a problem with the way we have been traversing trees: it breaks down the barrier between the client code (the application that uses the tree) and the provider code (the Tree implementation). Ideally, tree code should be general; it shouldn't know anything about expression trees. And the code that generates and traverses the expression tree shouldn't know about the implementation of the trees. This design criterion is called **object encapsulation** to distinguish it from the encapsulation we saw in Section 6.6, which we might call **method encapsulation**.

In the current version, the `Tree` code knows too much about the client. Instead, the `Tree` class should provide the general capability of traversing a tree in various ways. As it traverses, it should perform operations on each node that are specified by the client.

18.7 General tree traversal

In Pop11 natural way to separate tree traversal from operations on tree nodes is by using functional arguments: we simply pass a function which performs required operation as an argument to the traversal function and the traversal function applies operation to each node.

```
define traverse_preorder(tree, action);
    if tree == [] then
        return;
    endif;
    action(cargo(tree));
    traverse_preorder(left(tree), action);
    traverse_preorder(right(tree), action);
enddefine;
```

To get the same effect as our previous `print` function we can use the following function as the action:

```
define print_action(node);
    printf(cargo(node), '%p ');
enddefine;
```

Now `traverse_preorder(tree, print_action)` gives the same effect as `print(tree)`.

18.8 Simplified visitor pattern

There is another solution to the general traveral problem called **visitor pattern**. In simplified version of visitor pattern we will create a new abstract class, called `Visitable`. The items stored in a tree will be required to be visitable, which means that they define a method named `visit` that does whatever the client wants done to each node. That way the Tree can perform the traversal and the client can perform the node operations.

Here are the steps we have to perform to wedge an abstract class between a client and a provider:

1. Define an abstract class that specifies the methods the provider code will need to invoke on its components.
2. Write the provider code in terms of the new abstract class.
3. Define a concrete class that belongs to the abstract class and that implements the required methods as appropriate for the client.
4. Write the client code to use the new concrete class.

The next few sections demonstrate these steps.

18.9 Defining an abstract class

An abstract class definition looks a lot like a concrete class definition, except that it only specifies the interface of each method and not an implementation. The definition of `Visitable` is

```
define :mixin Visitable;
enddefine;
```

That's it! The word `mixin` is Pop11 keyword for an abstract class.

18.10 Implementing an abstract class

If we are using an expression tree to generate infix, then "visiting" a node means printing its contents. Since the contents of an expression tree are tokens, we'll create a new concrete class called `Token` that implements `Visitable`

```
define :class Token is Visitable;
    slot name;
enddefine;

define :method visit(t : Token);
    printf(str(t), '%p ');
enddefine;
```

The next step is to modify the parser to put `Token` objects into the tree instead of `Strings`. Here is a small example:

```
lvars char_rep = stingin('1 2 3 * +');
lvars item_rep = incharitem(char_rep);


lvars token = itemrep();
lvars tree = consTree (consToken(token), null, null);
```

This code takes the first token in the string and wraps it in a `Token` object, then puts the `Token` into a tree node. Now a methods can travesing the tree and invoke `visit` method on each node.

As an exercise, write a version of `printPreorder` called `visitPreorder` that traverses the tree and invokes `visit` on each node in preorder.

18.11 General visitor pattern

The simplified vistor pattern has one problem: each tree node has a single `visit` method, so can perform only single operation during traversal. In practice, we may need perform different operations, for example insted of printing nodes to the screen we may with to print them to disc, or simply count them.

Naive soultion would add multiple method to `Visitable` class, say `visit_print`, `visit_count`, etc. But then it looks that we also need multiple travesal functions (one for each visit method). In Pop11 our technique of passing function as parameter solves this problem: we just pass the method we need as a parameter. In fact, in Pop11 there is no need to insist that nodes are `Visitable` – we only need that nodes are correct arguments to visit method.

However, some languages (like Java) do not allow functions as parameter or make their use awkward. Then we need different solution: insted of a function we use objects of special `Visitor` class. `Vistor` must have a method for each kind of node it is going to visit. If parse tree contains tree kinds of tokens: `Number`s, `Operator`s and `Variable`s `Visitor` need methods `visit_Number`, `visit_Operator` and `visit_Variable` which take apropriate token as a parameter (`visit_Number` has a number as parameter).

We also need to modify `Visitable`: now it has `accept` method which takes `Vistor` object as parameter.

For example `accept` method for a `Number` looks like:

```
define :method accept(self : Number, visitor);
    visit_Number(visitor, self);
enddefine;
```

Now, the tree traversal functions instead of calling `visit` method call `accept` method. Moreover, tree traversal functions take visitor object as a parameter and pass it to the `accept` method.

18.12 Array implementation of trees

What does it mean to "implement" a tree? So far we have only seen one implementation of a tree, a linked data structure similar to a linked list. But there are other structures we would like to identify as trees. Anything that can perform the basic set of tree operations should be recognized as a tree.

So what are the tree operation? In other words, how do we define the Tree ADT?

constructor: Build an empty tree.

`empty`: Is this tree the empty tree?

`left`: Return the left child of this node, or an empty tree if there is none.

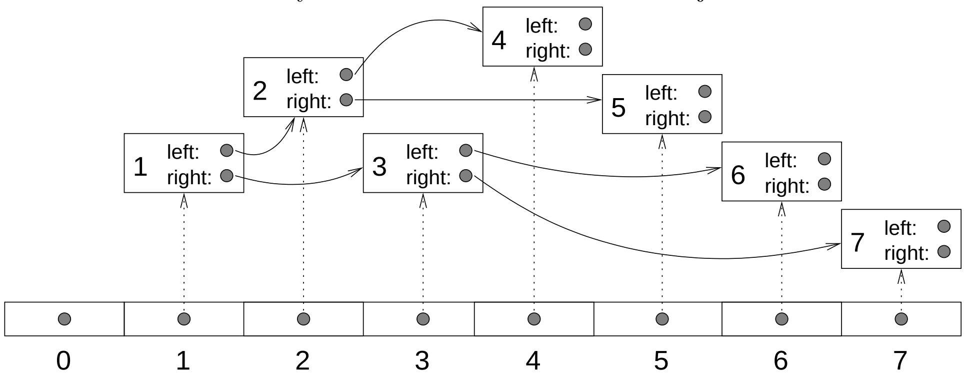
`right`: Return the left child of this node, or an empty tree if there is none.

`parent`: Return the parent of this node, or an empty tree if this node is the root.

In the implementation we have seen, the empty tree is represented by the special value `[]`. `left` and `right` are performed by accessing the instance variables of the node. We have not implemented `parent` yet (you might think about how to do it).

There is another implementation of trees that uses arrays and indices instead of objects and references. To see how it works, we will start by looking at a hybrid implementation that uses both arrays and objects.

This figure shows a tree like the ones we have been looking at, although it is laid out sideways, with the root at the left and the leaves on the right. At the bottom there is an array of references that refer to the objects in the trees.



In this tree the cargo of each node is the same as the array index of the node, but of course that is not true in general. You might notice that array index `1` refers to the root node and array index `0` is empty. The reason for that will become clear soon.

So now we have a tree where each node has a unique index. Furthermore, the indices have been assigned to the nodes according to a deliberate pattern, in order to achieve the following results:

1. The left child of the node with index i has index $2i$.
2. The right child of the node with index i has index $2i + 1$.
3. The parent of the node with index i has index $i/2$ (rounded down).

Using these formulas, we can implement `left`, `right` and `parent` just by doing arithmetic; we don't have to use the references at all!

Since we don't use the references, we can get rid of them, which means that what used to be a tree node is now just cargo and nothing else. That means we can implement the tree as an array of cargo objects; we don't need tree nodes at all.

Here's what one implementation looks like:

```
define :class Tree;
    slot buffer;
enddefine;

define make_Tree();
    return(consTree(initv(128)));
enddefine;
```

No surprises so far. The instance variable is a full vector. The constructor initializes this array with an arbitrary initial size (we can always resize it later).

To check whether a tree is empty, we check whether the root node is undefined. Again, the root node is located at index 1.

```
define :method empty(t : Tree);
    return (buffer(t)(1) == "undef");
enddefine;
```

The implementation of `left`, `right` and `parent` is just arithmetic:

```
define :method left(t : Tree, i);
    return (2*i);
enddefine;
define :method right(t : Tree, i);
    return (2*i + 1);
enddefine;

define :method parent(t : Tree, i);
    return (i div 2);
enddefine;
```

Only one problem remanins. The node "references" we have are not really references; they are integer indices. To access the cargo itself, we have to get or set an element of the array. For that kind of operation, it is often a good idea to provide methods that perform simple error checking before accessing the data structure.

```
define :method get_cargo(t : Tree, i);
    if i < 1 or i >= length(buffer(t)) then
        return([]);
    endif;
    return(buffer(t)(i));
```

```
enddefine;

define :method set_cargo(t : Tree, i, item);
    if i < 1 or i >= length(buffer(t)) then
        return;
    endif;
    item -> buffer(t)(i);
enddefine;
```

Methods like this are often called **accessor methods** because they provide access to a data structure (the ability to get and set elements) without letting the client see the details of the implementation.

Finally we are ready to build a tree. In another class (the client), we would write

```
lvars tree = make_Tree();
lvars root = 1;
set_cargo(tree, root, 'cargo for root');
```

The constructor builds an empty tree. In this case we assume that the client knows that the index of the root is `1` although it would be preferable for the tree implementation to provide that information. Anyway, invoking `setCargo` puts the string `"cargo for root"` into the root node.

To add children to the root node:

```
    set_cargo (tree, left(tree, root), 'cargo for left');
    set_cargo (tree, right(tree, root), 'cargo for right');
```

In the tree class we could provide a method that prints the contents of the tree in preorder.

```
define :method print_preorder(t : Tree, i)
    if get_cargo(t, i) = [] then
        return;
    endif;
    printf(get_cargo(t, i), '%p\n');
    print_preorder(t, left(t, i));
    print_preorder(t, right(t, i));
enddefine;
```

We invoke this method from the client by passing the root as a parameter.

```
print_preorder(tree, root);
```

The output is

```
cargo for root
cargo for left
cargo for right
```

This implementation provides the basic operations required to be a tree, but it leaves a lot to be desired. As I pointed out, we expect the client to have a lot of information about the implementation, and the interface the client sees, with indices and all, is not very pretty.

Also, we have the usual problem with array implementations, which is that the initial size of the array is arbitrary and it might have to be resized.

18.13 Glossary

binary tree: A tree in which each node refers to 0, 1, or 2 dependent nodes.

root: The top-most node in a tree, to which no other nodes refer.

leaf: A bottom-most node in a tree, which refers to no other nodes.

parent: The node that refers to a given node.

child: One of the nodes referred to by a node.

level: The set of nodes equidistant from the root.

prefix notation: A way of writing a mathematical expression with each operator appearing before its operands.

preorder: A way to traverse a tree, visiting each node before its children.

postorder: A way to traverse a tree, visiting the children of each node before the node itself.

inorder: A way to traverse a tree, visiting the left subtree, then the root, then the right subtree.

class variable: A variable (slot) shared by all objects of a class.

binary operator: An operator that takes two operands.

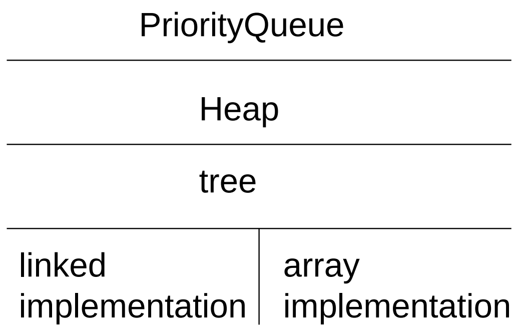
object encapsulation: The design goal of keeping the implementations of two objects as separate as possible. Neither class should have to know the details of the implementation of the other.

Chapter 19

Heap

19.1 The Heap

A heap is a special kind of tree that happens to be an efficient implementation of a priority queue. This figure shows the relationships among the data structures in this chapter.



Ordinarily we try to maintain as much distance as possible between an ADT and its implementation, but in the case of the Heap, this barrier breaks down a little. The reason is that we are interested in the performance of the operations we implement. For each implementation there are some operations that are easy to implement and efficient, and others that are clumsy and slow.

It turns out that the array implementation of a tree works particularly well as an implementation of a Heap. The operations the array performs well are exactly the operations we need to implement a Heap.

To understand this relationship, we will proceed in a few steps. First, we need to develop ways of comparing the performance of various implementations. Next, we will look at the operations Heaps perform. Finally, we will compare the Heap implementation of a Priority Queue to the others (arrays and lists) and see why the Heap is considered particularly efficient.

19.2 Performance analysis

When we compare algorithms, we would like to have a way to tell when one is faster than another, or takes less space, or uses less of some other resource. It is hard to answer those questions in detail, because the time and space used by an algorithm depend on the implementation of the algorithm, the particular problem being solved, and the hardware the program runs on.

The objective of this section is to develop a way of talking about performance that is independent of all of those things, and only depends on the algorithm itself. To start, we will focus on run time; later we will talk about other resources.

Our decisions are guided by a series of constraints:

1. First, the performance of an algorithm depends on the hardware it runs on, so we usually don't talk about run time in absolute terms like seconds. Instead, we usually count the number of abstract operations the algorithm performs.

2. Second, performance often depends on the particular problem we are trying to solve – some problems are easier than others. To compare algorithms, we usually focus on either the worst-case scenario or an average (or common) case.

3. Third, performance depends on the size of the problem (usually, but not always, the number of elements in a collection). We address this dependence explicitly by expressing run time as a function of problem size.

4. Finally, performance depends on details of the implementation like object allocation overhead and method invocation overhead. We usually ignore these details because they don't affect the rate at which the number of abstract operations increases with problem size.

To make this process more concrete, consider two algorithms we have already seen for sorting an array of integers. The first is **selection sort**, which we saw in Section 13.2. Here is the pseudocode we used there.

```
define selectionsort (array);
    for i from 1 to length(array) do
        ;;; find the lowest item at or to the right of i
        ;;; swap the ith item and the lowest item
    endfor;
enddefine;
```

To perform the operations specified in the pseudocode, we wrote helper methods named `find_lowest` and `swap`. In pseudocode, `find_lowest` looks like this

```
;;; find the index of the lowest item between
;;; i and the end of the array
```

```
define find_lowest(array, i);
    ;;; lowest contains the index of the lowest item so far
    lowest = i;
    lvars j;
    for j from i + 1 to length(array) do
        ;;; compare the jth item to the lowest item so far
        ;;; if the jth item is lower, replace lowest with j
    endfor;
    return(lowest);
enddefine;
```

And `swap` looks like this:

```
define swap (i, j);
    ;;; store a reference to the ith card in temp
    ;;; make the ith element of the array refer to the jth card
    ;;; make the jth element of the array refer to temp
enddefine;
```

To analyze the performance of this algorithm, the first step is to decide what operations to count. Obviously, the program does a lot of things: it increments `i`, compares it to the length of the deck, it searches for the largest element of the array, etc. It is not obvious what the right thing is to count.

It turns out that a good choice is the number of times we compare two items. Many other choices would yield the same result in the end, but this is easy to do and we will find that it allows us to compare most easily with other sort algorithms.

The next step is to define the "problem size." In this case it is natural to choose the size of the array, which we'll call n.

Finally, we would like to derive an expression that tells us how many abstract operations (specifically, comparisons) we have to do, as a function of n.

We start by analyzing the helper methods. `swap` copies several references, but it doesn't perform any comparisons, so we ignore the time spent performing swaps. `findi_lowest` starts at `i` and traverses the array, comparing each item to `lowest`. The number of items we look at is $n - i$, so the total number of comparisons is $n - i - 1$.

Next we consider how many times `findi_lowest` gets invoked and what the value of i is each time. The last time it is invoked, i is $n - 2$ so the number of comparisons is 1. The previous iteration performs 2 comparisons, and so on. During the first iteration, i is 0 and the number of comparisons is $n - 1$.

So the total number of comparisons is $1 + 2 + \cdots + n - 1$. This sum is equal to $n^2/2 - n/2$. To describe this algorithm, we would typically ignore the lower order term $(n/2)$ and say that the total amount of work is proportional to n^2. Since the leading order term is quadratic, we might also say that this algorithm is **quadratic time**.

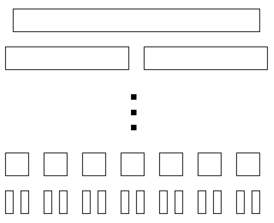
19.3 Analysis of mergesort

In Section 13.5 I claimed that mergesort takes time that is proportional to $n \log n$, but I didn't explain how or why. Now I will.

Again, we start by looking at pseudocode for the algorithm. For mergesort, it's

```
define merge_sort(array);
   ;;; find the midpoint of the array
   ;;; divide the array into two halves
   ;;; sort the halves recursively
   ;;; merge the two halves and return the result
enddefine;
```

At each level of the recursion, we split the array in half, make two recursive calls, and then merge the halves. Graphically, the process looks like this:



# arrays	items per array	# merges	comparisons per merge	total work
1	n	1	n−1	~n
2	n/2	2	n/2−1	~n
⋮	⋮	⋮	⋮	⋮
n/2	2	n/2	2−1	~n
n	1	0	0	

Each line in the diagram is a level of the recursion. At the top, a single array divides into two halves. At the bottom, n arrays (with one element each) are merged into $n/2$ arrays (with 2 elements each).

The first two columns of the table show the number of arrays at each level and the number of items in each array. The third column shows the number of merges that take place at each level of recursion. The next column is the one that takes the most thought: it shows the number of comparisons each merge performs.

If you look at the pseudocode (or your implementation) of merge, you should convince yourself that in the worst case it takes $m - 1$ comparisons, where m is the total number items being merged.

The next step is to multiply the number of merges at each level by the amount of work (comparisons) per merge. The result is the total work at each level. At this point we take advantage of a small trick. We know that in the end we are only interested in the leading-order term in the result, so we can go ahead and ignore the -1 term in the comparisons per merge. If we do that, then the total work at each level is simply n.

Next we need to know the number of levels as a function of n. Well, we start with an array of n items and divide it in half until it gets to 1. That's the same as starting at 1 and multiplying by 2 until we get to n. In other words, we want to know how many times we have to multiply 2 by itself before we get to n. The answer is that the number of levels, l, is the logarithm, base 2, of n.

Finally, we multiply the amount of work per level, n, by the number of levels, $\log_2 n$ to get $n \log_2 n$, as promised. There isn't a good name for this functional form; most of the time people just say, "en log en."

It might not be obvious at first that $n \log_2 n$ is better than n^2, but for large values of n, it is. As an exercise, write a program that prints $n \log_2 n$ and n^2 for a range of values of n.

19.4 Overhead

Performance analysis takes a lot of handwaving. First we ignored most of the operations the program performs and counted only comparisons. Then we decided to consider only worst case performance. During the analysis we took the liberty of rounding a few things off, and when we finished, we casually discarded the lower-order terms.

When we interpret the results of this analysis, we have to keep all this handwaving in mind. Because mergesort is $n \log_2 n$, we consider it a better algorithm than selection sort, but that doesn't mean that mergesort is *always* faster. It just means that eventually, if we sort bigger and bigger arrays, mergesort will win.

How long that takes depends on the details of the implementation, including the additional work, besides the comparisons we counted, that each algorithm performs. This extra work is sometimes called **overhead**. It doesn't affect the performance analysis, but it does affect the run time of the algorithm.

For example, our implementation of mergesort actually allocates subarrays before making the recursive calls and then lets them get garbage collected after they are merged. Looking again at the diagram of mergesort, we can see that the total amount of space that gets allocated is proportional to $n \log_2 n$, and the total number of objects that get allocated is about $2n$. All that allocating takes time.

Even so, it is most often true that a bad implementation of a good algorithm is better than a good implementation of a bad algorithm. The reason is that for large values of n the good algorithm is better and for small values of n it doesn't matter because both algorithms are good enough.

As an exercise, write a program that prints values of $1000 n \log_2 n$ and n^2 for a range of values of n. For what value of n are they equal?

19.5 Priority Queue implementations

In Chapter 17 we looked at an implementation of a Priority Queue based on an array. The items in the array are unsorted, so it is easy to add a new item (at the end), but harder to remove an item, because we have to search for the item with the highest priority.

An alternative is an implementation based on a sorted list. In this case when we insert a new item we traverse the list and put the new item in the right spot.

This implementation takes advantage of a property of lists, which is that it is easy to insert a new node into the middle. Similarly, removing the item with the highest priority is easy, provided that we keep it at the beginning of the list.

Performance analysis of these operations is straightforward. Adding an item to the end of an array or removing a node from the beginning of a list takes the same amount of time regardless of the number of items. So both operations are constant time.

Any time we traverse an array or list, performing a constant-time operation on each element, the run time is proportional to the number of items. Thus, removing something from the array and adding something to the list are both linear time.

So how long does it take to insert and then remove n items from a Priority Queue? For the array implementation, n insertions takes time proportional to n, but the removals take longer. The first removal has to traverse all n items; the second has to traverse $n - 1$, and so on, until the last removal, which only has to look at 1 item. Thus, the total time is $1 + 2 + \cdots + n$, which is (still) $n^2/2 - n/2$. So the total for the insertions and the removals is the sum of a linear function and a quadratic function. The leading term of the result is quadratic.

The analysis of the list implementation is similar. The first insertion doesn't require any traversal, but after that we have to traverse at least part of the list each time we insert a new item. In general we don't know how much of the list we will have to traverse, since it depends on the data and what order they are inserted, but we can assume that on average we have to traverse half of the list. Unfortunately, even traversing half of the list is still a linear operation.

So, once again, to insert and remove n items takes time proportional to n^2. Thus, based on this analysis we cannot say which implementation is better; both the array and the list yield quadratic run times.

If we implement a Priority Queue using a heap, we can perform both insertions and removals in time proportional to $\log n$. Thus the total time for n items is $n \log n$, which is better than n^2. That's why, at the beginning of the chapter, I said that a heap is a particularly efficient implementation of a Priority Queue.

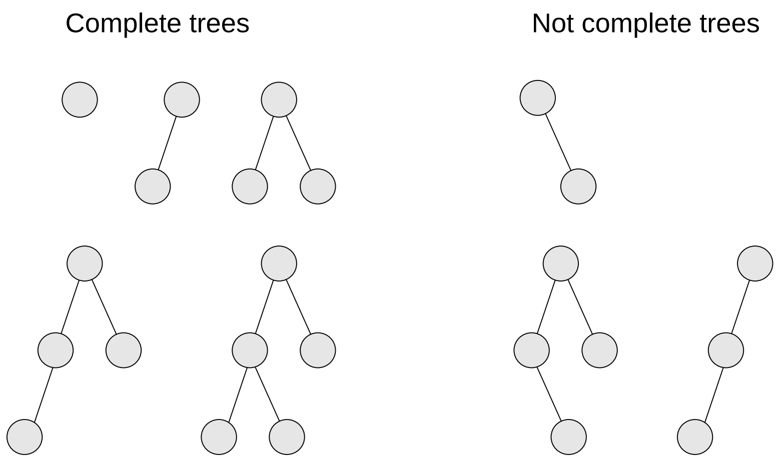
19.6 Definition of a Heap

A heap is a special kind of tree. It has two properties that are not generally true for other trees:

completeness: The tree is complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces.

heapness: The item in the tree with the highest priority is at the top of the tree, and the same is true for every subtree.

Both of these properties bear a little explaining. This figure shows a number of trees that are considered complete or not complete:



An empty tree is also considered complete.

Rigorous definition of completeness is somewhat tricky. Important role has height of subtrees. Recall that the **height** of a tree is the number of levels.

Starting at the root, if the tree is complete, then the height of the left subtree and the height of the right subtree should be equal, or the left subtree may be taller by one. In any other case, the tree cannot be complete.

One could think that if height of the left subtree and the height of the right subtree are equal and both subtrees are complete then the tree is complte. Unfortunatly, this is not the case – correct condition is more complicated.

In the simplest case the bottom level is completly filled. In other words all leaf elements of the tree are the the same level. We call such tree a full tree.

It is easy to see that in full tree height of the left subtree and the height of the right subtree are equal. Furthermore, if the tree is full, then both the left subtree and the right subtree are full. And, the converse is also true – so check that the tree is full we need to check both conditions.

It is natural to write these rules as a recursive function:

```
define is_full(tree);
    lvars left_height = height(left(tree));
    lvars right_height = height(right(tree));

    ;;; check the root node
    if left_height /= right_height then
        return(false);
    endif;

    ;;; check the children
    if not(is_full(left(tree))) then
        return(false);
    else
        return(is_full(right(tree)));
    endif;
```

```
enddefine;
```

What happens when the bottom level is not completely filled?

One case is when we are adding elements to the right subtree. Than, the height of the left subtree and the height of the right subtree should be equal, and the left subtree should be a full tree.

The next case is when the bottom level is filled in half—in this case the both left and right subtrees are full trees, but the left subtree is taller by one.

Finally, we may be adding elements to the left subtree—in this case the left subtree is taller by one and the right subtree is a full tree.

Furthermore, both subtrees are again complete trees – we need to check this.

Note that we can slightly simplify the rules: the last two cases are essentially the same: the left subtree is taller by one, the right subtree is a full tree and the left subtree is a complete tree. Similarly, in the first case of non-full tree (and the case of full tree) the height of the left subtree and the height of the right subtree should be equal, the left subtree should be a full tree, and the right subtree should be a complete tree.

Again we can write these rules as a recursive function.

```
define is_complete(tree);
    lvars left_height = height(left(tree));
    lvars right_height = height(right(tree));

    if left_height = right_height then
        if not(is_full(left(tree))) then
            return(false);
        else
            return(is_complete(right(tree)));
        endif;
    elseif eft_height = right_height + 1 then

        if not(is_full(right(tree))) then
            return(false);
        else
            return(is_complete(right(tree)));
        endif;
    endif;
enddefine;
```

For this example I used the linked implementation of a tree. As an exercise, write the same function for the array implementation. Also as an exercise, write the `height` function. The height of a empty tree (`[]`) is 0 and the height of a leaf node is 1.

Finally, our function computes multiple times heights of subtrees. You should think how to avoid this repeated computation.

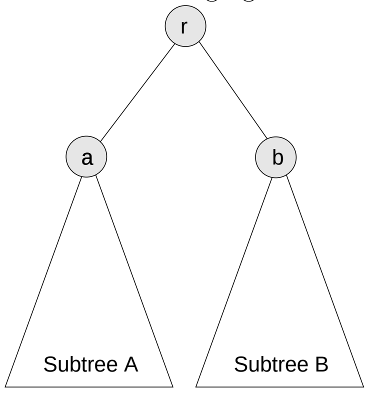
The **heap property** is recursive. In order for a tree to be a heap, the largest value in the tree has to be at the root, *and* the same has to be true for each subtree. As another exercise, write a method that checks whether a tree has the heap property.

19.7 Heap remove

It might seem odd that we are going to remove things from the heap before we insert any, but I think removal is easier to explain.

At first glance, we might think that removing an item from the heap is a constant time operation, since the item with the highest priority is always at the root. The problem is that once we remove the root node, we are left with something that is no longer a heap. Before we can return the result, we have to restore the heap property. We call this operation `reheapify`.

The situation is shown in the following figure:



The root node has priority `r` and two subtrees, A and B. The value at the root of Subtree A is `a` and the value at the root of Subtree B is `b`.

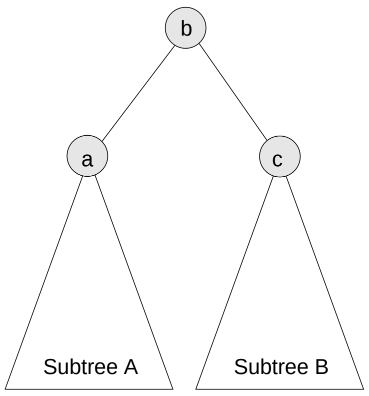
We assume that before we remove `r` from the tree, the tree is a heap. That implies that `r` is the largest value in the heap and that `a` and `b` are the largest values in their respective subtrees.

Once we remove `r`, we have to make the resulting tree a heap again. In other words we need to make sure it has the properties of completeness and heapness.

The best way to ensure completeness is to remove the bottom-most, right-most node, which we'll call `c` and put its value at the root. In a general tree implementation, we would have to traverse the tree to find this node, but in the array implementation, we can find it in constant time because it is always the last (non-null) element of the array.

Of course, the chances are that the last value is not the highest, so putting it at the root breaks the heapness property. Fortunately it is easy to restore. We know that the largest value in the heap is either `a` or `b`. Therefore we can select whichever is larger and swap it with the value at the root.

Arbitrarily, let's say that `b` is larger. Since we know it is the highest value left in the heap, we can put it at the root and put `c` at the top of Subtree B. Now the situation looks like this:



Again, `c` is the value we copied from the last entry in the array and `b` is the highest value left in the heap. Since we haven't changed Subtree A at all, we know that it is still a heap. The only problem is that we don't know if Subtree B is a heap, since we just stuck a (probably low) value at its root.

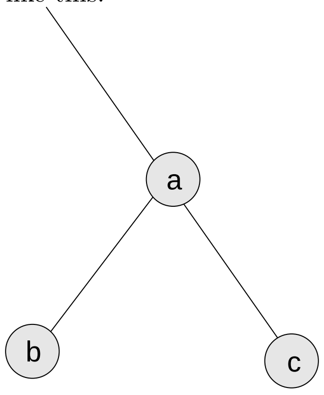
Wouldn't it be nice if we had a method that could `reheapify` Subtree B? Wait... we do!

19.8 Heap `insert`

Inserting a new item in a heap is a similar operation, except that instead of trickling a value down from the top, we trickle it up from the bottom.

Again, to guarantee completeness, we add the new element at the bottommost, rightmost position in the tree, which is the next available space in the array.

Then to restore the heap property, we compare the new value with its neighbors. The situation looks like this:



The new value is `c`. We can restore the heap property of this subtree by comparing `c` to `a`. If `c` is smaller, then the heap property is satisfied. If `c` is larger, then we swap `c` and `a`. The swap satisfies the heap property because we know that `c` must also be bigger than `b`, because `c > a` and `a > b`.

Now that the subtree is reheapified, we can work our way up the tree until we reach the root.

19.9 Performance of heaps

For both insert and remove, we perform a constant time operation to do the actual insertion and removal, but then we have to reheapify the tree. In one case we start at the root and work our way down, comparing items and then recursively reheapifying one of the subtrees. In the other case we start at a leaf and work our way up, again comparing elements at each level of the tree.

As usual, there are several operations we might want to count, like comparisons and swaps. Either choice would work; the real issue is the number of levels of the tree we examine and how much work we do at each level. In both cases we keep examining levels of the tree until we restore the heap property, which means we might only visit one, or in the worst case we might have to visit them all. Let's consider the worst case.

At each level, we perform only constant time operations like comparisons and swaps. So the total amount of work is proportional to the number of levels in the tree, a.k.a. the height.

So we might say that these operations are linear with respect to the height of the tree, but the "problem size" we are interested in is not height, it's the number of items in the heap.

As a function of n, the height of the tree is $\log_2 n$. This is not true for all trees, but it is true for complete trees. To see why, think of the number of nodes on each level of the tree. The first level contains 1, the second contains 2, the third contains 4, and so on. The ith level contains 2^i nodes, and the total number in all levels up to i is $2^i - 1$. In other words, $2^h = n$, which means that $h = \log_2 n$.

Thus, both insertion and removal take **logarithmic** time. To insert and remove n items takes time proportional to $n \log_2 n$.

19.10 Heapsort

The result of the previous section suggests yet another algorithm for sorting. Given n items, we insert them into a Heap and then remove them. Because of the Heap semantics, they come out in order. We have already shown that this algorithm, which is called **heapsort**, takes time proportional to $n \log_2 n$, which is better than selection sort and the same as mergesort.

As the value of n gets large, we expect heapsort to be faster than selection sort, but performance analysis gives us no way to know whether it will be faster than mergesort. We would say that the two algorithms have the same **order of growth** because they grow with the same functional form. Another way to say the same thing is that they belong to the same **complexity class**.

Complexity classes are sometimes written in “big-O notation”. For example, $\mathcal{O}(n^2)$, pronounced “oh of en squared” is the set of all functions that grow no faster than n^2 for large values of n. To say that an algorithm is $\mathcal{O}(n^2)$ is the same as saying that it is quadratic. The other complexity classes we have seen, in decreasing order of performance, are:

$\mathcal{O}(1)$	constant time
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	“en log en”
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(2^n)$	exponential

So far none of the algorithms we have looked at are **exponential**. For large values of n, these algorithms quickly become impractical. Nevertheless, the phrase “exponential growth” appears frequently in even non-technical language. It is frequently misused so I wanted to include its technical meaning.

People often use “exponential” to describe any curve that is increasing and accelerating (that is, one that has positive slope and curvature). Of course, there are many other curves that fit this description, including quadratic functions (and higher-order polynomials) and even functions as undramatic as $n \log n$. Most of these curves do not have the (often detrimental) explosive behavior of exponentials.

As an exercise, compare the behavior of $1000n^2$ and 2^n as the value of n increases.

19.11 Glossary

selection sort: The simple sorting algorithm in Section 13.2.

mergesort: A better sorting algorithm from Section 13.5.

heapsort: Yet another sorting algorithm.

complexity class: A set of algorithms whose performance (usually run time) has the same order of growth.

order of growth: A set of functions with the same leading-order term, and therefore the same qualitative behavior for large values of n.

overhead: Additional time or resources consumed by a programming performing operations other than the abstract operations considered in performance analysis.

Index