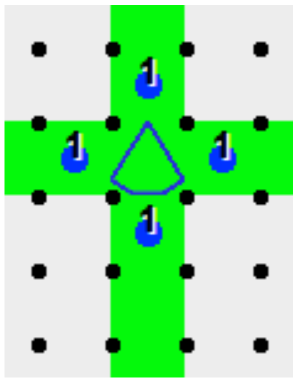
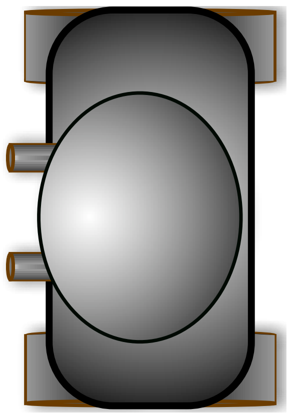
BeeperBot and PET





CMPUT 101 Course Notes

Joseph Culberson   Janelle Harms   Herb Yang

June 16, 2010

Contents

List of Tables

List of Figures

Preface

0.1 About this text

These notes are copyright under the Creative Commons license, obtained here `http://creativecommons.org/choose/`

*** NEED COPYRIGHT INFO, ALSO FOR BB AND PET ***

If you are viewing this on line, highlighted terms such as

`http://webdocs.cs.ualberta.ca/~joe/BB`
CMPUT 101 Source Site

are typically web links. Other highlighted items link to internal references, such as chapters 1, sections or figures 1.1. Depending on the software used to view this document, and on the dynamic nature of the web itself, these may or may not work.

These notes are an ongoing accumulation based on the CMPUT 101 course as it has evolved over the years. Please report typos and errors to `mailto:joe@cs.ualberta.ca`

0.2 Target Audience

This course is primarily targeted at students who are in disciplines other than Computing Science. No background or experience in computing is assumed. Although there is some low level programming using very simple languages, it is not intended to be a programming course, but rather a presentation of elementary computing science concepts.

0.3 Motivation

An underlying goal for this course is to present an introduction to computing science for non-CS students in a fashion analogous to an introductory physics course given to high school students. In an introductory high school physics course, as we recall them from many years ago, we might expect to see classical physics concepts such as force and mass and the relationships of acceleration, energy and work, for example. Elementary mechanical abstractions such as levers, pulleys and inclined planes, would be used to introduce these concepts. These latter entities might also be considered as elementary engineering abstractions, upon which all mechanical devices are built, but at the same time they are suitable for identifying and explaining some of the more abstract concepts in a tangible way.

Although modern physics might be more concerned with quantum mechanics and the Schrödinger equation, we would not expect a first level course to start there. Perhaps we overstate the case, or overdraw the analogy, but similarly we do not want to start from high level abstractions of computation and the latest software design techniques, but instead we want to identify the fundamental concepts and how they build up into more complex constructs through abstraction and composition, and the general principles of computation science. Although we will require some elementary abstractions of devices such as gates and circuits, we do not want to focus on the many engineering marvels of modern computers any more than an introductory physics course would focus on the engineering niceties of the latest Ferrari. Nor do we want to discuss the intricacies of managing a large software development project any more than an introductory chemistry course would discuss issues of managing a large chemical refinery.

In short, we want to identify, probe and discuss the analogs of wheels and pulleys, inclined planes, levers, forces and masses as we see them in computing science.

Also, note that this course is not a computer literacy course. We will not cover the use of a computer as a tool. For example, we will not discuss MS Word, Powerpoint or similar end user programs. In our view, using these tools as an introduction to computing science is akin to using driver education as an introduction to physics.

0.4 Why BB and PET?

Why do we use BeeperBot(BB) and Program Exploration Tool(PET) instead of teaching a language like Java? The goal of this course is to provide insight into the concepts underlying computation, not to make programmers out of you. BB can be used to present many of the fundamental concepts of computing without the complexity of a full blown modern programming language. PET is a simple programming language with structure similar to the language C, and the advantage of a visual emulator that facilitates understanding of how the language works in terms of the underlying computational model.

Most modern programming languages are very complex. They are designed for production level work and cannot be fully grasped in a single term, especially by anyone not even familiar with elementary concepts. Here is a quote found on the internet

To make matters worse, there are more syntax rules in Java than *there are in English, and the error messages you get from the compiler are often not very helpful.*

OOPWeb.com `http://www.oopweb.com/Java/Documents/ThinkCSJav/Volume/chap01.htm` Chapter 1: The way of the program.

Syntax rules govern the formation of statements in a programming language. BB and PET have far fewer syntax rules, and are much easier to learn. The PET tool used later allows a more flexible development of algorithmic concepts, many of which carry directly over to complete programming languages. As an analogy, in an introductory cooking course you would expect to focus more on the fundamentals of ingredients and techniques than on the mechanics of the latest multipurpose cooking range. Believing that learning a more complex programming language will make you cognizant of computing science is akin to believing that learning to run a more complex electric range will make you a great chef.

Nevertheless, much of what you learn here will carry over to modern programming.

0.5 Study Hints for this Course

Since this course is largely targeted at non-science and non-mathematics students, a word on how to use these notes may be in order. This material is best approached by "doing", not memorization. Intense study means working lots of exercises, not reviewing masses of reading material.

A good analogy is to consider a sport such as hockey. No matter how many books you may read, nor how many lectures you may attend, unless you put on skates, and grab a stick and practice, you will never learn to play hockey.

When reading sections of these notes about BeeperBot or PET, you should have a computer handy with the tool readily at hand to try each idea as it is presented.

Chapter 1

Introductory Concepts Using BeeperBot

In this chapter we introduce some fundamental concepts of computing science, using the program BeeperBot. Once you obtain BeeperBot, it would be a good idea to have it running while reading the rest of the chapter, so that you can check out the features such as the extensive Help menu. Also, use BeeperBot to check out the examples, and to try variations as you read.

1.1 What is BeeperBot?

BeeperBot, or BB for short, is a programming environment to control a primitive graphical robot that exists in a two dimensional grid world. We often refer to the robot as BB itself. The robot can only interact with walls and beepers. For walls, BB can only detect them, and if she tries to move through a wall, an error occurs.

BB can create, destroy, collect and distribute beepers. She has no built-in way to count, and so BB can only distinguish between having one or more beepers, or none.

Nevertheless, BB can carry out powerful computations, given enough time. Many elementary but fundamental concepts of computing science can be illustrated easily with BB. And that is of course the goal of this software.

1.2 Obtaining BeeperBot (BB)

Before proceeding further, you need to obtain the program BeeperBot from the website, if you have not already done so. (We assume you have access to a computer for this course. If you do not, then the following exercises will have to be carried out in one of the computing science labs. We suggest you be prepared to spend quite a bit of time in the labs in this case.)

1. Download the file `beeper.jar` from this site.

   `http://webdocs.cs.ualberta.ca/~joe/BB`

2. Put the file `beeper.jar` in the folder where you keep applications.

To uninstall BeeperBot, simply remove the file `beeperbot.jar`. In addition, if you have changed your settings, in your home directory there may be file `.gvrsettings` which you may also remove.

1.3 Getting Started with BB

On a MAC or PC you should be able to start BB by clicking or double clicking the file `beeper.jar`.

On Unix or Linux enter the command

```
java -jar /Applications/beeperbot.jar
```

where of course you must replace `Applications` with the path to the location of `beeperbot.jar` on your system. You can also start it this way in a terminal in MAC OS X if you prefer that to clicking.

On MAC OS X you can also drag `beeperbot.jar` to the dock (at the document end) and start it from there.

After starting BB, you should have a window that looks like Figure 1.1. The graphics may vary slightly depending on the system you are using. This may also differ a bit if you have used BB before and saved a different set of preferences. The following assumes this is the first time you have run BB, and that BB is using the default settings.

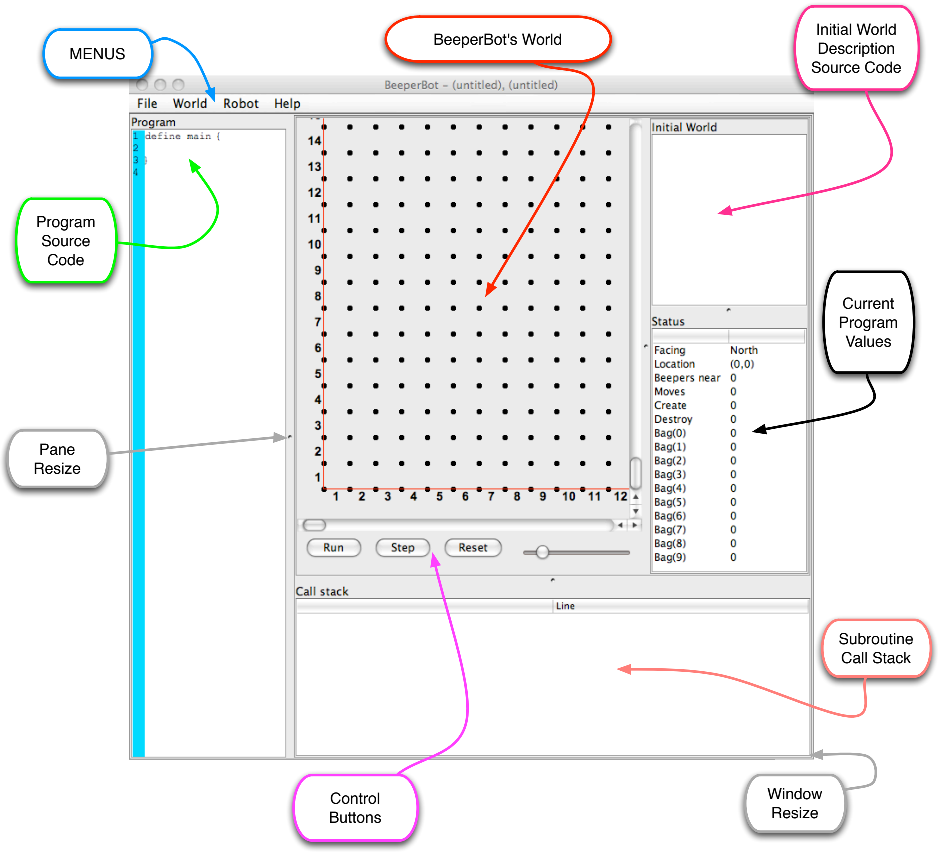
The window consists of five panes which we now examine, assuming you have BB running and will use this as a tutorial.

Program Pane: This is found on the left side of the window, and is an editable text area. When started it contains a default program consisting of only the lines shown in Figure 1.1.

Try it now: Click in the window and enter the text " *move*" at line 2 followed by the line " *pick_beeper*" at line 3. Your program should now look like

```
define main {
  move
  pick_beeper
}
```

Initial World Pane: Located on the top right of the window, this is another editable text window. Click on the initial world pane and enter the lines below exactly as shown (note the capital 'N', all other letters are lower case).



```
define main {
}
```

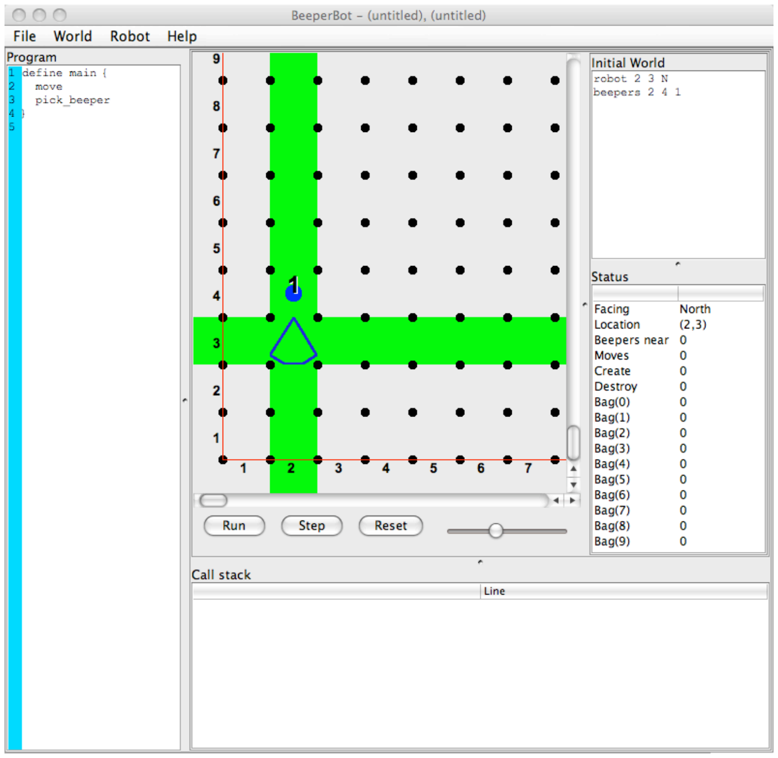
Facing	North
Location	(0,0)
Beepers near	0
Moves	0
Create	0
Destroy	0
Bag(0)	0
Bag(1)	0
Bag(2)	0
Bag(3)	0
Bag(4)	0
Bag(5)	0
Bag(6)	0
Bag(7)	0
Bag(8)	0
Bag(9)	0

Figure 1.1: On initial start up, BeeperBot will look like this, without the colorful labels of course. Your display may differ slightly depending on the system you are using and the options you are using.

```
robot 2 3 N
beepers 2 4 1
```

World Pane: BeeperBot's World is located in the center top of the window. At the bottom of this pane there are three buttons that control execution of the program. There is also a slider knob that changes the scale of the world display.

- **Reset** Initializes the world. **Try it now**. Click the **Reset** button. You will see that the world changes to display the location of the robot, with a blue shape representing the robot, and a green cross indicating the coordinates. (The green coordinate display can be turned off in the menu **Robot** if desired.) After clicking **Reset** the Window should look like this.



```
define main {
  move
  pick_beeper
}
```

```
robot 2 3 N
beepers 2 4 1
```

Facing	North
Location	(2,3)
Beepers near	0
Moves	0
Create	0
Destroy	0
Bag(0)	0
Bag(1)	0
Bag(2)	0
Bag(3)	0
Bag(4)	0
Bag(5)	0
Bag(6)	0
Bag(7)	0
Bag(8)	0
Bag(9)	0

- **Step** Executes one step of the program each time it is clicked. **Try it now**. Note that each time you click, a line of text is highlighted in the program pane. The highlighted line is the next step of the program to be executed.

  Note that on the first click, the line *define main*{ is highlighted, the second click highlights the line " *move*". On the third click the line " *pick_beeper*" is highlighted, and at the same time the robot in the world pane moves one step forward.

If you keep clicking the program will start over again, after first resetting to the initial world description.

- **Run** Executes the program. While the program is running this button turns to **Pause**, although for this short program it may be too quick to notice. If the program is paused, then clicking **Run** again continues the program from where it was paused. Clicking **Reset** at any time stops the program and resets it to the initial world conditions.

Status Pane: This is located on the right center. It is a display only text window. It indicates the following:

- **Facing** The direction the robot is currently facing, with "North" being "up".
- **Location** The column and row BB is currently in.
- **Beepers Near** The number of beepers in the same location as BB.
- **Moves** The total number of *move* and *turn_left* statements that have been executed so far in the current run.
- **Create** How many beepers BB has created so far in the current run.
- **Destroy** How many beepers BB has destroyed so far in the current run.
- **Bag(0)...Bag(9)** In Auxiliary mode BB has 10 bags, labeled 0 through 9. Each bag will be listed on a separate line. The number on the right indicates the number of beepers currently in the bag. Note: in primitive mode BB has no bags, and in standard only one bag. The display will change according to which mode BB is in.

Call Stack Pane This is a text display window that shows the current subroutine calls.

Tutorial Exercise

Using the program and world defined above, step through the program one step at a time, and observe precisely when the robot moves and when it picks up a beeper putting it into *bag(0)*. After each click of the **Step** button, carefully check both the World display, and the information in the **Status** pane. Notice in particular that the code line is highlighted *before* the action takes place. Also note that some code lines do not change either the world or status file.

1.4 A brief overview of BB's menus

As indicated in Figure 1.1 there are 4 menu items in BeeperBot.

File For loading, saving and creating new programs, as displayed and edited in the **Program** pane. Also contains the item **Exit** which exits BeeperBot.

World For loading, saving and creating new initial world descriptions, as displayed and edited in the **Initial World** pane.

Robot This item has has several functionalities discussed separately below. Broadly, it has controls for running the program, and sets various preferences.

Help Links to a built-in manual describing the interface, the program syntax and listing the commands and conditionals available in the different operating modes.

The manual under the **Help** menu might be sufficient to learn BeeperBot. Note however, as with any computer language, there is a huge gap between learning the language elements and learning to effectively program in that language.

For the **File** and **World** menus, all files are saved as plain text, and can be viewed using most editors. However, you should not edit program or world files using editors such as Microsoft Word as these introduce formatting text that may cause BeeperBot to fail. Be sure to choose distinct names for your World and Program files: e.g. *exercise_1_world.txt* and *exercise_1_program.txt* so that they will not overwrite each other.

The Robot Menu

As stated above, the **Robot** menu has several purposes. Here we list the various items with a brief description of each.

Run

Step

Reset These have the same functionality as the corresponding buttons in the **World Display** pane described above; namely they run the current program.

0 - Slow

1 - Medium

2 - Fast

3 - Full Speed These buttons control the speed at which the code executes. **Slow** allows the user to follow the execution of most programs, but is very tedious for longer running pieces of code. **Fast** is probably too fast to follow what is happening with both the code and the robot in real time, but can give a view of what bits of code are being used the most. **Full Speed** does not update the display and does not trace the execution of the code, but instead runs the code until the program halts, and then displays the result.

Note that execution speed can also be modified by statements in the source program. This feature is handy for working with some of the larger programs in the notes.

Highlight coordinates This toggles on or off the green cross bars indicating the row and column where the robot is located in the **World Display** pane.

Show beeper dots This toggles whether or not a dot is displayed in the cells containing one or more beepers. If it is toggled off, then only the number of beepers is displayed in the cell.

Use large fonts This toggle affects the size of the font used for the text in the **Program**, **Initial World** and **Status** panes. The actual size used is set in the pop-up window activated by the **Settings** item in the **Robot** menu. This may be useful when displaying the program in a classroom setting for example.

Use large beeper dots When toggled on, instead of displaying a dot, this fills the entire cell with an ugly pinkish color. This is useful when trying to use BeeperBot to create simple graphic images. This toggle has no effect if **Show beeper dots** is turned off.

Settings Pops up a window with the following contents

Calls before... A text window that requires a number. The default is 5000, and this means that for long running programs every 5000 steps an alert window will pop up, asking if you wish to continue the program. This is useful if for example you are running a program at **Full Speed** and it happens to have an infinite loop. It is expected that most BB programs will not run this long.

Operating Mode with radio buttons **Primitive Mode, Standard Mode** and **Auxiliary Mode**. Primarily, these control the number of beeper bags that BB can use. In primitive mode, BB has no bags, in standard mode she has 1 bag, and in Auxiliary mode BB has 10 bags. See the manual under the **Help** menu for more information.

Large Font Size Use the scroll to select the font size to use when **Use large fonts** is toggled on. This does not affect the default font size.

Save After making settings, click this button. Note: until you click this save button, BeeperBot will not remember recently seen files, nor the toggles that are set in the **Robot** menu.

1.5 Programming BeeperBot

Before proceeding further do the following.

Start BeeperBot.

If BB does not display 10 bags in the **Status** pane, then do the following.

1. Under the **Robot** menu, scroll down and click **settings**
2. In the pop-up window, click the **Auxiliary** radio button.
3. Also in the pop-up window, click the **save** button.

Throughout these notes, unless stated otherwise, we will assume you are using Auxiliary mode. BeeperBot will remember to start in Auxiliary mode on subsequent startups once you save the settings as described.

1.5.1 Coding an Initial World

Figure 1.2 illustrates how the world file constructs the initial world for BeeperBot to start in.

Notice that there is already a wall running along the west side of the world, and another along the bottom edge. There are no comparable walls on the east or north, unless you specify them. Basically, BB can go as far east or north as it wishes, until your computer runs out of memory.

To be sure you understand, you should practice making a few arbitrary worlds using BeeperBot.

Tutorial Exercises

1. What happens if you try to make two (or more) robots, using two robot command lines?
2. What happens if you use the line

   ```
   robot 1 6 N 55
   ```

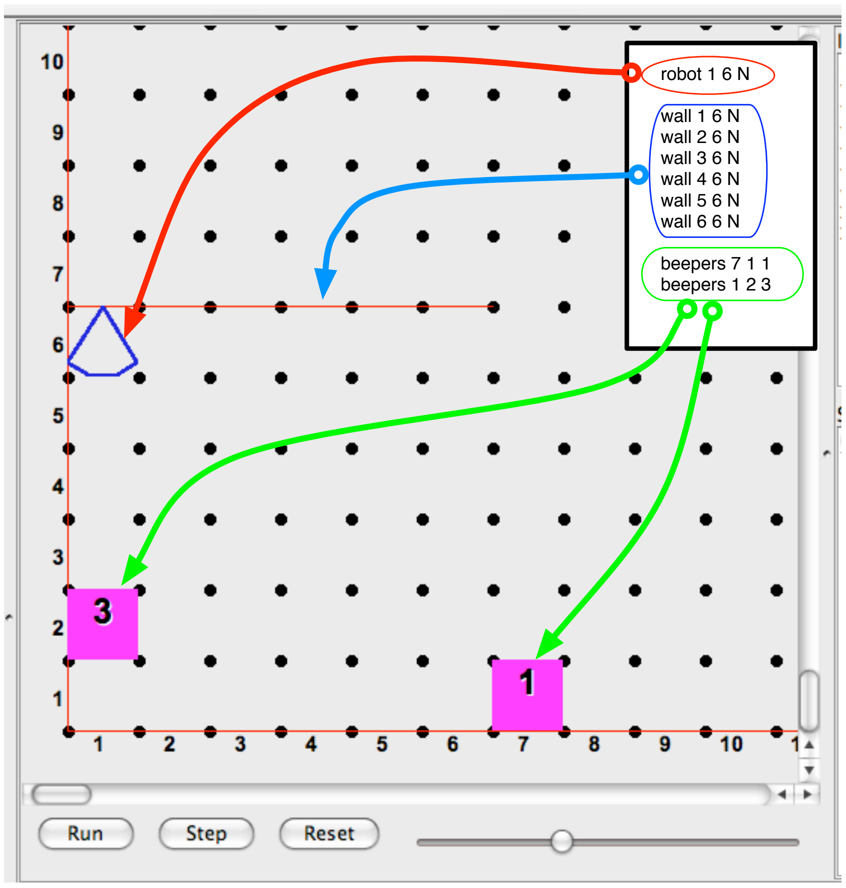
   to define the robot location? This is a legal way to define the initial location of the robot, but your task is to find out what the number 55 at the end of the line does.

1.5.2 Programming Language

While reading this part, you should have BeeperBot running with the manual in the **Help** menu item **Contents**. The **Help** menu item **Contents** has a more extensive description of the commands than is found here. This section focuses as much on using the BeeperBot tool and general programming ideas as details of the language.

First let us summarize the statements available in BeeperBot.

Here we organize the command list slightly differently. There are two general types of statements, *commands* that cause some action to happen and *conditionals* which test some condition and evaluate to **true** or **false** depending on whether the condition holds or not.



```
robot 1 6 N
wall 1 6 N
wall 2 6 N
wall 3 6 N
wall 4 6 N
wall 5 6 N
wall 6 6 N
beepers 7 1 1
beepers 1 2 3
```

Figure 1.2: How the world file works: The robot command indicates the initial location of the robot, the two numbers indicating the column and row, the N indicating it should be facing north. The six wall commands place walls on the north side of row six, one in each column from 1 to 6. The other option is to place the walls on the east side of a cell using 'E'. The beepers command can place any number of beepers on any cell in the initial world.

Robot Control

These statements allow you to program movement of the robot, while avoiding walls.

Movement Commands *move* and *turn_left*.

Wall Checking Conditionals *front_is_clear*, *not_front_is_clear*, and similar for left and right

Compass Checking Conditionals *facing_north*, *not_facing_north*, and similar for east, south and west.

Working with Beepers

These statements allow you manipulate beepers, both in the world and in the beeper bags. Remember, BB has 10 beeper bags (in **auxiliary** mode) each of which can store any number of beepers. A cell in the world may also hold any number of beepers.

Manipulation Commands *create_beeper*, *destroy_beeper*, *pick_beeper*, *put_beeper*, *move_beeper(0,1)*... *move_beeper(9,0)*

Location Check Conditionals *next_to_a_beeper*, *not_ next_to_a_beeper*

Bag Check Conditionals *has_beeper(0)* ... *has_beeper(9)*, and *not_has_beeper(0)* ... *not_has_beeper(9)*.
(Note: *any_beeper_in_beeper_bag* and its negation are equivalent to *has_beeper(0)* and its negation, and are included only for compatibility reasons.)

Program Control

These statements control the order in which the code is executed during a process. Note that *process* refers to the running of the code.

Code Control Statements *if*, *if ... else*, *do*, *while*

Subprogram Definition *define*

Program Speed *set_speed*, *restore_speed*

Comments any line that begins with a # is treated as a comment, and ignored when the program is running.

We believe that the best way to become familiar with the concepts of running a program is to try a few examples. Check the BeeperBot **Help** menu for what these commands do.

1.6 Coding Concepts and Terminology

Let us consider the following source code to establish some terminology that will be used throughout this course. The program draws a square when executed. It will be referred to again later in this chapter.

```
# Example Program CMPUT 101 Lec 1&2 Culberson

define main {
  do (4) {
     do (3) {
        create_beeper
        move
        # what happens if the next
        # turn_right is moved here
     }
     turn_right
  }
}

define turn_right {
  turn_left
  turn_left
  turn_left
}
```

Here is some terminology.

Source Code

The example program above is an example of *source code.* It is the description of the program, written to follow the rules of the programming language.. Frequently the language requires a text document as in this example. However, a few languages use a graphical form, such as that used in the language Scratch `http://scratch.mit.edu/` and some low level languages, or machine languages, are basically numbers. We will see a little bit of that later in the course.

Syntax and Statements

Many online dictionaries define *syntax* as the study of the rules used to govern the construction of grammatical sentences. For a programming language such as that used by BeeperBot the *statements* are quite restricted in the form they take, because the syntax rules are mathematically precise and must be followed precisely. The reason for this is that computers are inherently non-intutitive, and even the slightest error will cause BB to give up.

The syntax rules for BB are given in the `Contents` section of the `Help` menu. For BB, each line consists of exactly one *statement.* For this purpose, you can think of blank lines, comment lines and lines containing only “ } ” as *null* statements.

Code Block

In BeeperBot source code, the statements between braces are referred to as *a code block*. Thus, lines 5–10 form a code block, and lines 4–12 form another code block, which contains the first code block as a *nested block* or *sub-block*. In fact lines 3–13 and lines 15–19 are two more blocks.

Process

As stated previously, a *process* refers to the act of executing the code. For our purposes you can think of a process as being the actions of the computer when you hit the **Run** or **Step** buttons. Alternatively you can simulate the process by carefully following the steps laid out in the program by hand. This hand process we refer to as *tracing* the execution of the code.

Subroutine or Procedure

The source code in lines 15–19 is a special code block in that it defines a *subroutine* or *procedure*. When the program is executing and reaches line 12, we say that the *turn_right* subroutine is *called*. Confusingly, we may also refer to the turn_right statement in 12 as a *call* even when we are talking about the code when not running.

Loops

These **do** statements are examples of a general construction called *loops*, since a process repeats the statements iteratively the required number of times. Later we will see how to build *conditional loops* using a **while** statement.

Abstraction

The *turn_right* routine is an example of *abstraction*. In computing science, *abstraction* generally means that we want to think of the external or over all effects of a piece of code versus the details of its implementation. In this regard

- the external property of the routine is to cause the robot to turn 90^o to the right
- the implementation causes the robot to make 3 left turns.

1.7 Tracing Execution

One of the skills a programmer must develop is the ability to hand simulate a process described by a program. We refer to this as *tracing* the execution of a program.

We will trace the execution of the following program. We include statement numbers here as they would appear in BB for easy reference to the statements.

```
1 define main {
2    move
3    while (next_to_a_beeper){
4           pick_beeper
5           turn_left
6    }
7    while (has_beeper(0)) {
8         create_beeper
9         move_beeper(0,1)
10       move
11  }
12 }
```

We assume the following initial world description.

```
robot 3 3 N
beepers 3 4 2
```

What do we need to do? We need to track the following:

1. What statement we are about to execute next in the source code.
2. Where the robot is located
3. What BB has in her bags
4. What the world looks like.

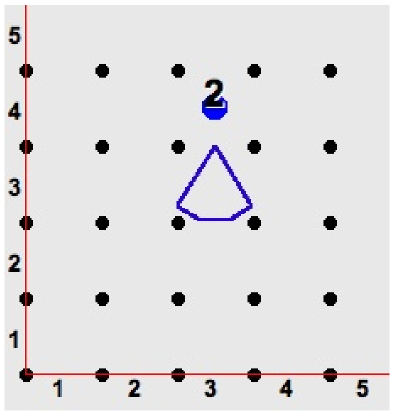
The first three items on our list we can do in a simple table. The last item requires some sort of diagram. For this text we will use images copied from executing BB on this program. But remember, when doing it by hand, you will need to make a diagram that you can update.

For this first example, we will do a very detailed accounting. As you gain experience, you will learn to abstract chunks of code and perform aggregate steps at once. For now, you should run BB as you read this, executing each step, or sequence of steps, as it is discussed.

Where do we start? First we do a reset. Then let us simulate the program from the time of the first click of the **Step** button. Since only bags 0 and 1 are referenced in the code, we only need to track their contents in the table. The table below indicates the relevant data after this first click. At this point the line "*1 define main* {" is highlighted. Note that it has not executed. In particular note that there are no beepers next to BB, that is in cell (3,3).

Process Step	Statement Number	Robot Location	Beepers Near	Number of Moves	Contents of	
					Bag(0)	Bag(1)
1	1	(3,3),N	0	0	0	0

At this point BB's world looks like this



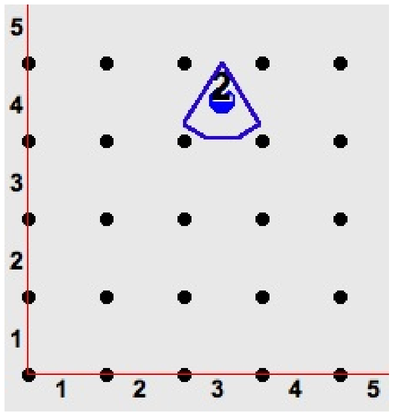
If we click once more, we are ready to execute statement 2, the *move* statement. Nothing else has changed.

Process Step	Statement Number	Robot Location	Beepers Near	Number of Moves	Contents of Bag(0)	Contents of Bag(1)
1	1	(3,3),N	0	0	0	0
2	2	(3,3),N	0	0	0	0

On the next click BB moves one step north to location (3,4), and is now standing next to the 2 beepers. Note that the number of moves has increased to 1.

Process Step	Statement Number	Robot Location	Beepers Near	Number of Moves	Contents of Bag(0)	Contents of Bag(1)
1	1	(3,3),N	0	0	0	0
2	2	(3,3),N	0	0	0	0
3	3	(3,4),N	2	1	0	0

At this point BB's world looks like this

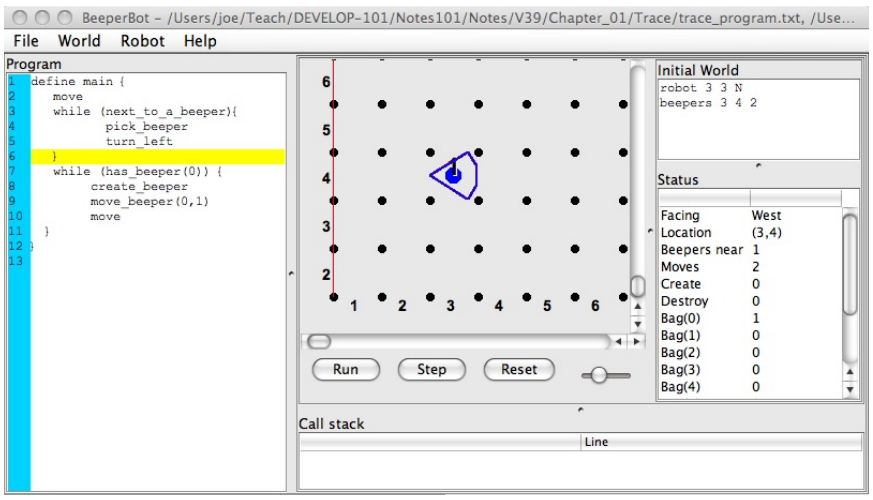


In the source code, statement 3, "*while (next_to_a_beeper)*{", is highlighted, and if you are following this in BB as you should be, you will notice that there is a green check mark at the end of the source code line. This indicates that the condition in the while loop currently evaluates to **true**. In this case this means there is indeed one or more beepers in the cell where BB is located.

Since the condition is **true** the next step will start executing the body of the while loop, the *body* being the statements inside the code block. After another two more clicks the trace table should look like this:

Process Step	Statement Number	Robot Location	Beepers Near	Number of Moves	Contents of Bag(0)	Bag(1)
1	1	(3,3),N	0	0	0	0
2	2	(3,3),N	0	0	0	0
3	3	(3,4),N	2	1	0	0
4	4	(3,4),N	2	1	0	0
5	5	(3,4),N	1	1	1	0
6	6	(3,4),W	1	2	1	0

We show here the entire BB window so that you can check with the run you are doing using BB and the table above.


BeeperBot – /Users/joe/Teach/DEVELOP-101/Notes101/Notes/V39/Chapter_01/Trace/trace_program.txt, /Use...
File World Robot Help
Program
Run
Step
Reset
Call stack
Line

```
1  define main {
2     move
3     while (next_to_a_beeper){
4            pick_beeper
5            turn_left
6     }
7     while (has_beeper(0)) {
8           create_beeper
9           move_beeper(0,1)
10          move
11    }
12 }
13
```

Initial World

```
robot 3 3 N
beepers 3 4 2
```

Status

Facing	West
Location	(3,4)
Beepers near	1
Moves	2
Create	0
Destroy	0
Bag(0)	1
Bag(1)	0
Bag(2)	0
Bag(3)	0
Bag(4)	0

Note that the number of moves has increased to 2. This is because a *turn_left* is counted as a move by the BB program.

At this point, the next click on the **Step** button causes the code execution to move back to statement 3. The *while* condition needs to be tested again. Since there is still one beeper next to BB, the body of the *while* will execute again. Here is what the trace table should look like after a few more clicks.

Process Step	Statement Number	Robot Location	Beepers Near	Number of Moves	Contents of	
					Bag(0)	Bag(1)
1	1	(3,3),N	0	0	0	0
2	2	(3,3),N	0	0	0	0
3	3	(3,4),N	2	1	0	0
4	4	(3,4),N	2	1	0	0
5	5	(3,4),N	1	1	1	0
6	6	(3,4),W	1	2	1	0
7	3	(3,4),W	1	2	1	0
8	4	(3,4),W	1	2	1	0
9	5	(3,4),W	0	2	2	0
10	6	(3,4),S	0	3	2	0

Notice that BB is now facing south, has made a total of three “moves” and has two beepers in Bag(0). There are no beepers left in the world. These are the key things you need to know about the process so far.

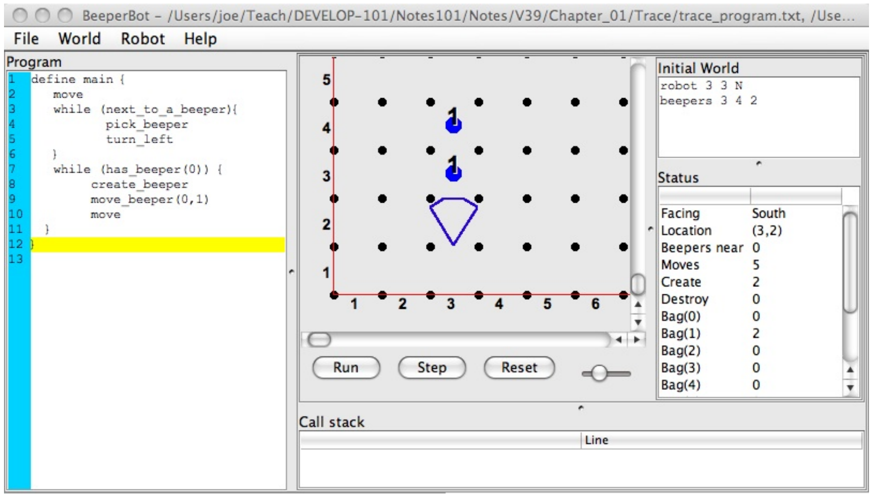
On the next step the source code process will once more go back to the statement 3 to check the *while* condition, but this time there are no beepers left in the room, so the condition is **false**. The next step will therefore jump out of the while and the process will continue at statement 7, the second *while* in the program.

Now let us present the remainder of the process at a slightly more abstract level. Note however that you should continue to trace the execution step by step and verify the table entries below.

Each time around this *while* loop, BB first creates a beeper in the cell it is standing in. Then she moves one beeper from Bag(0) to Bag(1).The purpose of doing this is so that eventually the process will stop looping. After moving the beeper, BB moves one step forward.

You should be able to see now that BB will create two beepers, one in cell (3,4) and one in cell (3,3) and stops in location (3,2). We present here the complete tracing table and the final result.

Process Step	Statement Number	Robot Location	Beepers Near	Number of Moves	Contents of Bag(0)	Contents of Bag(1)
1	1	(3,3),N	0	0	0	0
2	2	(3,3),N	0	0	0	0
3	3	(3,4),N	2	1	0	0
4	4	(3,4),N	2	1	0	0
5	5	(3,4),N	1	1	1	0
6	6	(3,4),W	1	2	1	0
7	3	(3,4),W	1	2	1	0
8	4	(3,4),W	1	2	1	0
9	5	(3,4),W	0	2	2	0
10	6	(3,4),S	0	3	2	0
11	3	(3,4),S	0	3	2	0
12	7	(3,4),S	0	3	2	0
13	8	(3,4),S	0	3	2	0
14	9	(3,4),S	1	3	2	0
15	10	(3,4),S	1	3	1	1
16	11	(3,3),S	0	4	1	1
17	7	(3,3),S	0	4	1	1
13	8	(3,3),S	0	4	1	1
14	9	(3,3),S	1	4	1	1
15	10	(3,3),S	1	4	0	2
16	11	(3,2),S	0	5	0	2
17	7	(3,2),S	0	5	0	2
16	12	(3,2),S	0	5	0	2


BeeperBot – /Users/joe/Teach/DEVELOP-101/Notes101/Notes/V39/Chapter_01/Trace/trace_program.txt, /Use...
File
World
Robot
Help
Program
Run
Step
Reset
Call stack
Line

```
1  define main {
2      move
3      while (next_to_a_beeper){
4              pick_beeper
5              turn_left
6      }
7      while (has_beeper(0)) {
8             create_beeper
9             move_beeper(0,1)
10            move
11     }
12 }
13
```

Initial World

```
robot 3 3 N
beepers 3 4 2
```

Status

```
Facing         South
Location       (3,2)
Beepers near   0
Moves          5
Create         2
Destroy        0
Bag(0)         0
Bag(1)         2
Bag(2)         0
Bag(3)         0
Bag(4)         0
```

There are several comments we should make here. This trace was more detailed than we would generally expect. With a little thought, you can see that tracing all the values over the lines that do not change values is probably wasted effort, and can in fact lead to increased errors. Instead you should focus on the statements that actually change some value such as the contents of a cell, BB's location or the contents of one or more bags.

In this example we did not trace the number of beepers created, although that could easily be done. Tracing the location of BB in the table is probably more error prone and difficult than simply working on a diagram for most cases.

In the end, the level of detail you use to trace a program will be a matter of personal preference. On an exam you may be asked to trace the actions of BeeperBot given a program and starting with a particular world. While writing exams you will not have access to BeeperBot. Thus, you must learn to trace the actions of a program out by hand, predicting what BB will do. While practicing examples, after you have traced out the example by hand, and only after, you should run the program in BB and see if the result agrees with your prediction. If not, work through the example in more detail until you understand where you went wrong. The exam questions will typically be mainly marked on the correctness of the final world description.

Remember, BB is always right — in the sense that she always does exactly what she is programmed to do. Unfortunately programs do not always do what is expected. Tracing is one way of determining if the program has errors, and what they are.

1.8 Two Sample BeeperBot Programs

In order to show how BeeperBot is programmed, we present here two small programs. Text versions of these programs can be found in a Sample Programs at `http://webdocs.cs.ualberta.ca/~joe/BB`, ready to be opened in BeeperBot after downloading.

Example One

Here is a simple program to outline a square using beepers. Note we include line numbers for reference, they are not part of the code. This part goes in the program pane, if you are typing it in.

```
 1 # Example Program CMPUT 101 Lec 1&2 Culberson
 2
 3 define main {
 4   do (4) {
 5      do (3) {
 6         create_beeper
 7         move
 8         # what happens if the next
 9         # turn_right is moved here
10      }
11      turn_right
12   }
}

define turn_right {
  turn_left
  turn_left
  turn_left
}
```

For the initial world, type in

```
robot 6 6 N
```

Starting at location (6,6) will ensure that BB does not run into a wall.

You should run this code to see what it does, and trace the code by hand to be clear you understand each part.

Things to note: There are two activities going on when you run a BeeperBot program. First, the software in the BeeperBot tool is running a *process* defined by your program source code. Evidence of this activity can be observed in the sequence of yellow highlights of the source code.

Second, the robot is moving around its world, manipulating beepers. This activity is controlled by the execution of your program. You can see evidence of this activity by observing the movements of the robot, and watching the changing values in the **Status** pane.

In order to understand programming, you must come to grips with these two notions of process, and how they relate to one another, or more specifically, how the processing of your code controls the actions of the robot.

When running the program, also note the iteration counter associated with each *do* in the **Program** pane. Check carefully how the counters are tested on the completion of each iteration. In particular, follow this code until you clearly understand how the nesting works. Take the hint from the comments, and change the program to see how it changes the behavior of the robot when the *turn_right* command is moved inside the inner loop.

Example Two

```
# Second Example in the notes
# draws a spiral length depending the number in initial pile
define get_all {
    while (next_to_a_beeper) {
        pick_beeper
    }
}

define make_side {
    while (has_beeper(0)) {
        move_beeper(0,1)
        create_beeper
        move
    }
}

define restore_bag {
    while (has_beeper(1)) {
        move_beeper(1,0)
    }
    # reduce the number by one
    put_beeper
    destroy_beeper
}

define main {
    get_all
    while (has_beeper(0)){
        make_side
        restore_bag
        turn_left
    }

}
```

This program can use the following initial world description.

```
robot 7 7 E
beepers 7 7 10
```

This program draws a spiral. You may wish to turn on the beeper dots option in the **Robot** menu to get a better visual effect.

Note that in this program we have chosen to define the various subroutines before the *define main* line. Nevertheless, the program knows to start at the correct line, since it is *main*.

Note the use of beeper bags 0 and 1 to keep track of how many beepers are left. We use the create_beeper command to make copies, scattering them along one per location.

You are invited to try your hand at making other designs using BB.

1.9 Example Program Development

So far we have presented some programs for you to read and follow. Now the task gets more difficult. Here we outline the steps required to develop a program for a particular task. Note that in general this is a much more difficult task than learning the programming language, because it calls for a special kind of creativity. Like most creative endeavors, it is difficult to learn by any method other than practice.

Here is the problem. The initial world consists of BeeperBot being located at an arbitrary spot in the world, facing in an arbitrary direction. In the location 1 1 there is a pile of an arbitrary number of beepers. BB must move to the location 1 1, pick up all the beepers and take them back to its initial location, and put them down. Or equivalently, create a pile of equal size.

Here are eight example initial worlds.

robot 7 8 E beepers 1 1 7	robot 13 2 W beepers 1 1 2	robot 1 3 N beepers 1 1 15	robot 1 1 E beepers 1 1 0
robot 200 356 E beepers 1 1 750	robot 21 1 S beepers 1 1 3	robot 19 19 N beepers 1 1 19	robot 1 1 E beepers 1 1 10

Algorithms and Generality: *Algorithms* are just general solutions to computation problems that we can easily encode in some language. For our purposes, for now, we can think of an algorithm as a BeeperBot program. In fact, in the remainder of this chapter we will use the terms interchangeably.

One of the key ideas in algorithm design is the notion of making the algorithm *general*. A non-general approach to the above problem for example would be to consider the top left initial world and then write a program that first turns BB twice to the left, moves 7 times, turns left again, moves 8 times, then a *do* loop that picks up 7 beepers, then turns left again, then moves 7 times, turns left again, moves 8 times, then drop eight beepers.

This program would of course not work at all for any of the other 7 examples, and it would not work for any of the millions of other possible initial worlds. Thus this program is *not general.*

Whenever we ask for a program or algorithm to solve some problem, we mean that it must work for any instance that fits the general description given. But this requirement adds significantly to the difficulty of designing the program.

Before going further, it is a good idea to get a high level description of the major tasks that we will need to have BB perform.

1. BB must find the corner at location 1 1.
2. BB must pick up the beepers
3. BB must return to the starting location
4. BB must put down the correct number of beepers

Starting with such a high level task description is referred to as *top down design* in computing science. Using this high level description as a guide, we can write some BB code as follows.

```
define main {
  go_to_corner
  pick_all_beepers
  return_to_start
  drop_all_beepers
}
```

Note however, that we cannot execute this code because none of the instructions we have used are part of BB's language! But not to worry! Because we have the capability of writing subroutines in BB, all we have to do is write a subroutine for each of these four tasks and we are done. Easy!

So, lets start with the first, namely *go_to_corner*. Now we look at our table of instances and we see some problems. Uh oh. First BB can be at any location, even, as in the two examples on the right side of the table starting at location (1,1). Or BB can be very far away as in the bottom left example of the table. One thing we have going for us is that the cell (1,1) is located in the corner formed by the two boundary walls. So if we follow along one of the walls until we reach the corner, we will have solved that issue.

But we also notice that initially BB may be facing in any of the four directions. Here with a careful look at the **Help/Contents** menu item we will find that there are tests such as *facing_west* that we can use to test which direction BB is facing. We refer to these tests collectively as BB's compass.

With these tools in hand, we can begin to code the first task. First, we will use the compass to get oriented towards one of these walls, namely the west wall. This is an arbitrary choice; we could equally choose to find the south wall first. We will then keep moving until BB hits a wall, then turn to the south and keep moving until BB hits a wall. We can tell when BB comes to a wall using the condition *front_is_clear* or its negation.

So, following our top down idea, lets write the subroutine *go_to_corner*. Again, we see four subtasks that we must accomplish to do this.

```
define go_to_corner {
   turn_west
   go_to_wall
   turn_south
   go_to_wall
}
```

But, BB cannot execute this code either because these are also not commands in the language BB knows. Hmm? Are we actually making progress?

Yes we are. What we must do is keep refining our task descriptions until we get to a level that is in fact just the commands that are built into the language. But how do we know we are getting there? Like any creative task, we have no idea how to communicate this other than to say keep practicing writing programs until it becomes clear to you. We will complete this program as an example, which may help.

It turns out that the next level of refinement will allow us to complete each of the subtasks in the subroutine *go_to_corner*. First, to turn west, we just keep turning left as long as we are not facing west. Here is that routine.

```
define turn_west {
   while (not_facing_west) {
     turn_left
   }
}
```

A very similar routine will do for *turn_south*, and we leave that as an exercise.

To run up to a wall, the following will do.

```
define go_to_wall {
   while (front_is_clear) {
     move
   }
}
```

So, this completes the first top level task of getting to the corner. When designing a program with many subtasks like this, it is usually a good idea to see if the parts are really working as we go along. Since we only have code for the *go_to_corner* subroutine, we must comment out or delete the other routines from *main* procedure. Once this is done, we can test that BB will find the corner any starting position. Here is what your test procedure should look like, except that of course you still have to supply the code for the *turn_south* subroutine.

```
define main {
  go_to_corner
  # pick_all_beepers
  # return_to_start
  # drop_all_beepers
}

define go_to_corner {
   turn_west
   go_to_wall
   turn_south
   go_to_wall
}

define turn_west {
   while (not_facing_west) {
     turn_left
   }
}
```

```
define turn_south {
   # EXERCISE
}

define go_to_wall {
   while (front_is_clear) {
     move
   }
}
```

Try it and verify that it works.

Sticking with our main agenda, we next have to write the subroutine for main task number 2, the subroutine *pick_all_beepers*. This is also left as a straight forward exercise for the reader. Once you have this, you can add it to your test program, delete the comment symbol from the line in the main procedure, and verify that BB works thus far.

Now we come to the third main task. And almost immediately we see that we have a problem. Since we are solving the general problem, we have to return to some arbitrary location, but if we are running a process based on our current source code, we have no idea where that location might be!

Clearly we must somehow modify the source code for the first step and create a *memory* so BB will know how to return her initial position. If we could give BB some twine, then she could roll out the twine as she moved to the corner, and follow it back after she picked up the beepers. BB cannot deal with actual twine, but we can use beepers to record the trail. We could try a number of ways. One way is to have BB lay out beepers as she moves to the corner, then follow them back again. This can be made to work, but it is a bit tricky. We leave this method as a difficult exercise. Be careful of special cases, for example what do you do if BB starts out on cell (1,1)?

Instead our approach will be to add beepers to two different beeper bags; one bag will record the number of moves south BB makes in getting to the corner, and the other the number to the west. To return she simply moves as far north as she recorded moving south, and then as far east as she recorded going west. It turns out that making this general solution using the bags is not so hard.

First we must modify the subroutine *go_to_corner* so that it records the moves in each of the two directions. We notice that in this routine the actual moves are made in the subroutine *go_to_wall* and that this routine is used twice, once to find the west wall and again to find the south wall. Since we want to record two different numbers, we could replace this with two different routines, but instead we will do the following.

First we modify *go_to_wall* to record the number of moves in Bag(0).

```
define go_to_wall {
   while (front_is_clear) {
     move
     create_beeper
     pick_beeper
   }
}
```

We cannot leave these beepers in Bag(0) because they would get mixed up with the beepers from the next move sequence and the ones we pick up in the corner. So we modify the subroutine *go_to_corner* as follows

```
define go_to_corner {
   turn_west
   go_to_wall
   move_all_1
   turn_south
   go_to_wall
   move_all_2
}
```

and add the routine

```
define move_all_1 {
   while(has_beeper(0)) {
     move_beeper(0,1)
   }
}
```

We leave the routine *move_all_2* which should move all beepers from Bag(0) to Bag(2) as another exercise.

Note that we have used Bag(1) to record the number of moves west BB made, and so it will be the number of moves east BB must make to get back to her starting position. Bag(2) records the number of moves south BB made to get to the corner, and so it will be how many moves north BB must make to return.

So now we are ready to write the routine *return_to_start*. The general idea is fairly straight forward. We first turn north and then move the number of times recorded in Bag(2). We then turn east and move the number of times recorded in Bag(1).

```
define return_to_start {
   turn_north
   move_north
   turn_east
   move_east
}
```

We leave *turn_north*, *turn_east* and *move_east* as exercises. Here is the code for *move_north*. Note that we use Bag(9) as a *trash can* to hold the beepers used for counting moves. If desired, you could move these to Bag(0) then put them down and then destroy them, leaving Bag(9) empty when finished.

```
define move_north {
  while (has_beeper(2)) {
     move_beeper(2,9)
     move
  }
}
```

You should complete the exercises and thus the entire program and try it out on a few of the example initial worlds.

Here is one last exercise. How would you modify the program if it was also required that when BB returns to her starting location she must also turn to face in the same direction she was facing in the initial world?

Naming Subroutines

It should be apparent that you are free to pick your subroutine names. Basically, a name has to start with a letter, and then can be any sequence of letters, numbers or under-bars that you care to make it. The name you choose is up to you, and has no impact on BeeperBot. For example, in this program you could change the name *go_to_corner* to *joe_needs_coffee* and provided the change is made consistently in the main procedure and in the define statement, the program will work just the same. Try it if you do not believe me.

However, you will likely agree that for human readers of your program, *go_to_corner* will be more meaningful.

Never forget that a TA will be marking your code, so you should always strive to choose names that are as helpful as possible. If the TA is unable to understand your name choices, you will likely lose marks.

Along this same line of thinking, it is always a good idea to add comments to explain your routines and code. Again, obscure uncommented code will lose marks, even if it seems to work correctly.

1.10 BeeperBot and the Concept of State

The two purposes of this section are to (i) introduce some fundamental terminology used in computing science, and (ii) to present yet another view of how a process proceeds when a BeeperBot program executes.

All classical digital computation is founded on the concept that a system (computer, or a component of a computer) is in some discrete *state* at a given time and then can make a *transition* into a different state at a later time. In modern computers this transition is typically prompted by an *input* from some event, and the new state is completely determined by the current state and the event. During the transition, the system may also generate some output.

To make this more concrete, consider a very common household example, an electric light switch. Typically, such switches have two discrete states, *on* and *off*. Suppose a switch is initially *off*, in which case we say it is in its *initial* state. Then someone flips the switch. This is an *input event.* As a result, the switch is now in the *on* state. The *transition* is from 'off' to 'on', and as an output (effect) the light comes on. As computations go, this may seem rather trivial, but as we will see later in the course, collections of millions of such two state devices are the foundation of every modern digital computer.

We use the word *discrete* when referring to state here to distinguish from continuous. A dimmer switch may have an almost infinite variation of settings, and so might be considered continuous. We are not interested in continuous systems for this course (although some early non-digital computers used them.)

Examples of state machines abound in the typical household. For example, consider the timer in a typical automatic washing machine, somewhat simplified. It may have states for fill, wash, drain, rinse, spin and stop. Once the user sets the knob to fill, the initial state, the washer fills with water. When it senses sufficient water has been let in, it switches to the wash state, which causes an output of rocking the tub. After a certain elapsed time, the clock sends an input to switch states and the tub will drain. Then it fills again, rocks the tub again in the rinse step, switches to the drain state again, then the spin state and halts.

This idea of stepping through states under precise, deterministic control, is at the heart of all digital computing. (We ignore bleeding edge ideas such as quantum computing. Quantum states are a more complex concept. Similarly, we are ignoring non-deterministic computing, parallel computing, biological computers, etc. as these require more sophisticated models of computation. We note however that all these models have a notion of state.)

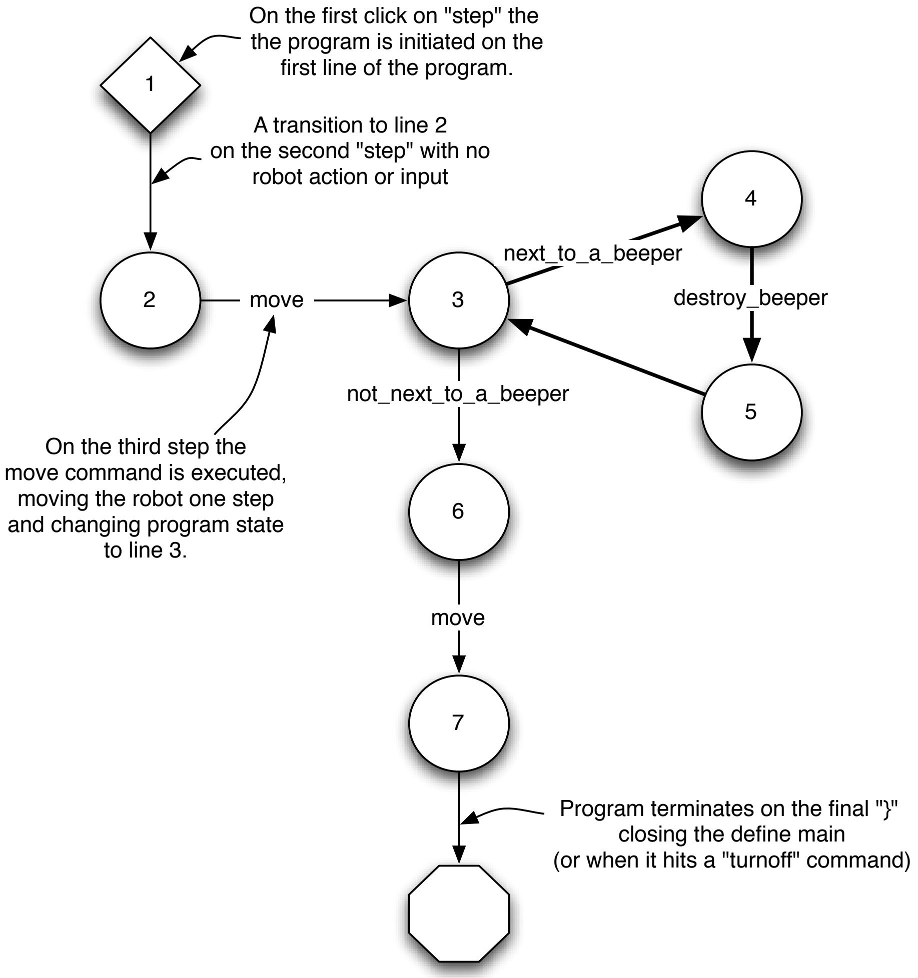
In BeeperBot when we have a program with no subroutines, we identify each line number (except blank lines) as the label of a distinct *state.* (We discuss how to deal with subroutines in the next subsection.) Since any program has a finite set of lines of code, this means there are only a *finite number of states* associated with such a program. A *transition* occurs when we step from one line of code to another, that is we transition from one state to another. If the line of code is not a *do, while* or *if* statement then there is only one possible next state that the transition can lead to.

If the source line is a *if* or *while* statement, then there will be two choices for the next state, one when the condition is **true** and one when it is **false**. In this case, the *event* triggering the transition consists of the line of code together with the value of the conditional. This value in turn depends on the configuration of the world at the time the statement is executed.

The key idea is to understand that in a computation as we discuss it, the sequence of actions is uniquely and precisely determined by the states, code and configuration of the world.

There has to be a starting point, which we call the *initial state.* In BeeperBot this is always the line containing the text *define main* {. Similarly, the *initial*

```
Program                                          World
1 define main {                                  robot 3 3 N
2     move                                       beepers 3 4 2
3     while (next_to_a_beeper) {
4         destroy_beeper
5     }
6     move
7 }
```



On the first click on "step" the the program is initiated on the first line of the program.
1
A transition to line 2 on the second "step" with no robot action or input
4
next_to_a_beeper
2
move
3
destroy_beeper
not_next_to_a_beeper
5
On the third step the move command is executed, moving the robot one step and changing program state to line 3.
6
move
7
Program terminates on the final "}" closing the define main (or when it hits a "turnoff" command)

Figure 1.3: State diagram containing a *while* loop

configuration is described by the code in the **Initial World** pane.

We can diagram a BeeperBot program in a straightforward way. Such a diagram is called a *state diagram* or a *flow chart.* This may assist you in understanding the process associated with a simple program, and in tracing such a process. Our first example is shown in figure 1.3.

It may be useful to follow this discussion using BeeperBot using the **Step** button. First click **Reset**. The first step after reset highlights the define main line, which in this program is line 1. This corresponds to the initial state of the state diagram, which is shaped like a diamond, and has label 1. The next step transitions to line 2, the *move* statement. This corresponds very naturally to state 2 in the diagram. The move instruction has not yet been executed.

On the next step, the program transitions to state 3, corresponding to line 3, and during the transition the robot moves one step. We can say that this transition has an output, which modifies the world configuration by moving the robot to a new location.

In state 3 there are two possible transitions, depending on whether or not the robot is next to a beeper in the world. If the robot is next to a beeper, then the next transition is to state 4. Otherwise, it is to state 6.

Now consider the three states labeled 3, 4 and 5. In the diagram these form a cycle, $3 \rightarrow 4 \rightarrow 5 \rightarrow 3$, where the arrows indicating transitions in the figure are highlighted. The only way the process can get out of this cycle is that eventually there is no beeper next to the robot. Fortunately, the transition from state 4 to state 5 has the effect of destroying one beeper in the world in the cell next to BB. Since this happens every iteration of the cycle, eventually no beepers will be left in the cell. It should be apparent that if there are two beepers next to BB, then the process must iterate this cycle two times before it can transition to state 6. If instead the initial world had placed ten beepers in this cell, then the process would iterate ten times.

As a final note, there are three types of transitions in the state diagram:

1. Action or output transitions; e.g. *move, destroy_beeper, turn_left* etc.

2. Conditional or input transitions; *while* and *if.* There are always exactly two transitions out of any state corresponding to a conditional. It is possible to model the *do* statement in a way similar to the *while*, by inventing a counter. We leave this as something to think about in one of the exercises.

3. Null (epsilon) transitions; transitions on lines of code such closing braces, define statements etc. These do not interact with the world, and are simply there to make the diagrams complete.

In the next two sections we explore some further insights that can be garnered from these diagrams.

Program

```
1 define main {
2     move
3     if (next_to_a_beeper) {
4             destroy_beeper
5     }
6     move
7 }
```

World

```
robot 3 3 N
beepers 3 4 2
```

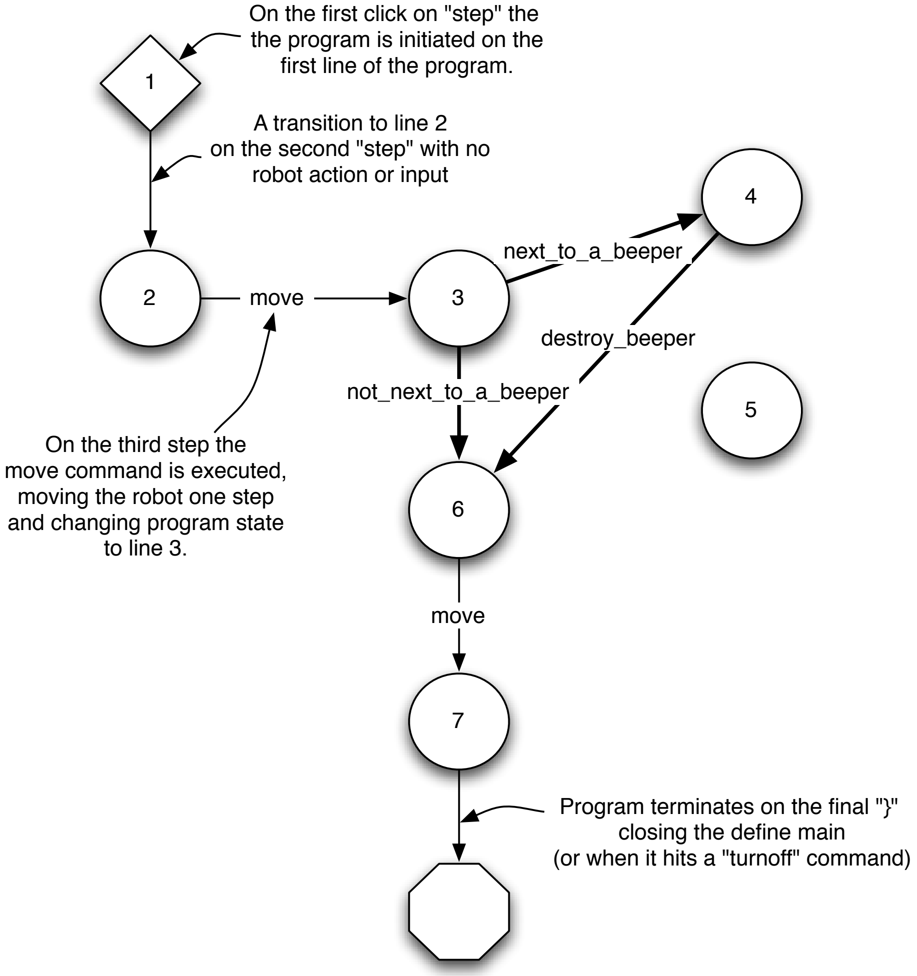

On the first click on "step" the the program is initiated on the first line of the program.
1
A transition to line 2 on the second "step" with no robot action or input
2
move
3
4
next_to_a_beeper
destroy_beeper
5
not_next_to_a_beeper
On the third step the move command is executed, moving the robot one step and changing program state to line 3.
6
move
7
Program terminates on the final "}" closing the define main (or when it hits a "turnoff" command)

Figure 1.4: State diagram on an *if* statement.

1.10.1 Illustrating the Difference between *if* and *while*

We frequently find first time students get confused between *if* and *while* statements, sometimes using one when they need the other. In figure 1.4 we illustrate code that is identical to that in Figure 1.3 except that the *while* has been replaced by *if*. Note the difference this makes in the state diagram. Now there is no cycle on states 3, 4 and 5. (Line 5 never gets highlighted by the process.)

If 1 or more beepers are placed in cell (3,4) in the initial world (two are placed in the illustration) then when the program is in state 3, the transition to state 4 follows. From state 4, there is a transition to state 6, which has the effect of destroying one beeper. But from state 6 there is no transition back to state 3, so the other beeper is never destroyed. If ten beepers are initially placed, still only one is destroyed, the other nine escape without injury.

If no beepers are placed initially, then the transition from state 3 is directly to state 6, so no attempt to destroy (the non-existent) beeper is made. This is good, because an attempt to destroy a beeper that is not there will cause a run time error.

1.10.2 Subroutines, Abstraction and State Diagrams

It is usually possible to rewrite a BB program containing subroutines so that is has no subroutines. The idea is to simply to copy the code in the subroutine and replace the line that calls it. Of course, if there are several lines calling the routine, then each of them has to be expanded this way. And if the original subroutine called other subroutines, then the process has to be repeated until there are no calling statements left. From a coding point of view, this is generally a bad idea. The program can rapidly become hard to understand.

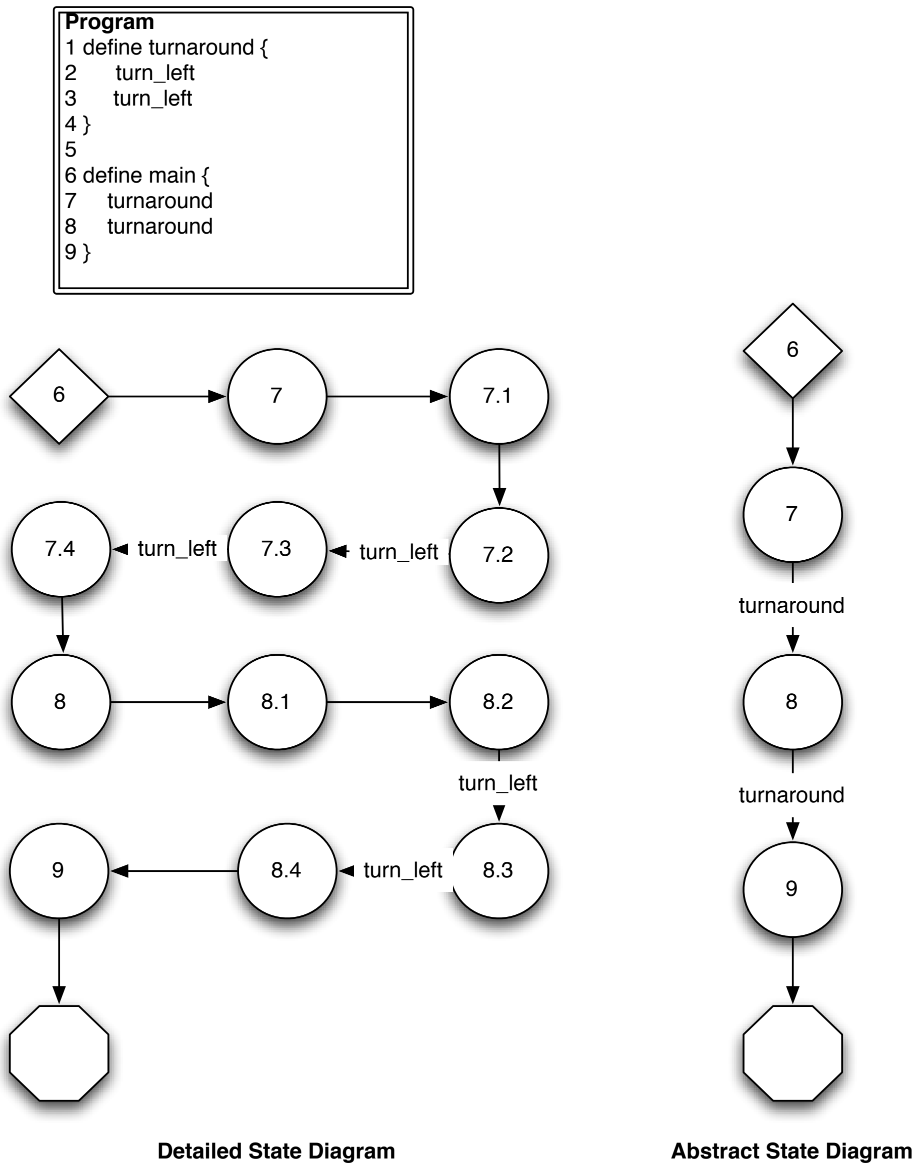
However, we show how to use this idea to come up with state labels for a program that has subroutines. And we illustrate why it pays to use abstraction. Figure 1.5 shows two ways to use state diagrams when subroutines are allowed.

Note that in this example the initial state is 6, since line 6 is the *define main* line.

The idea we use on the left side of the figure is to label the state by two numbers separated by a dot. The first number represents the calling line, the second indicates the particular line in the subroutine. If the subroutine in turn calls another subroutine, then we would add more dots and so on. As you can see this gets messy, and we had to fold the diagram up like a snake to make it fit.

But on the right hand side of Figure 1.5 we make use of the power of abstraction. Here we include only the states corresponding to lines in the *main* procedure. We label the arrows representing transitions with the name of the subroutine indicating the *external* effect of the subroutine, and ignore the details of how the subroutine is implemented. We refer to this as an **Abstract State Diagram**.

If we wish, we could draw a separate diagram for the procedure *turnaround*. Its initial state would be labeled 1, and its last state would be 4. We leave this



```
Program
1 define turnaround {
2      turn_left
3      turn_left
4 }
5
6 define main {
7     turnaround
8     turnaround
9 }
```

Figure 1.5: BeeperBot State Example Two

as an exercise for the reader.

1.11 Sample BeeperBot Questions

The following are sample questions for BeeperBot. Some of these questions are easier than others. An exam will typically have a similar mix. Note that on an exam you will not be able to run BeeperBot to check your answers so it is important that you can do these on your own. However, if you are struggling or unsure of your answer, fire up BB and try it!

1.11.1 Tracing Questions

1. Given this World: *(Recall that the 4 at the end of the robot command puts 4 beepers in bag(0).)*

   ```
   robot 5 5 N 4
   ```

   and this Program:

   ```
   do (6) {
      if (has_beeper(0)) {
             put_beeper
             move
       }  else {
              turn_left
       }
    }
   turn_off
   ```

   (a) Draw a state diagram corresponding to this diagram. How do you deal with with the loop created by the **do(6)** program line?

   (b) When executed, and BB turns off

      i. What direction is BB facing?

      ii. Where is BB located?

      iii. How many beepers remain in BB's beeper bag 0?

2. World:

   ```
   robot 5 5 N 0
   beepers 5 5 1
   beepers 5 6 1
   beepers 5 7 1
   beepers 5 8 1
   beepers 5 9 1
   ```

   Program:

```
while (next_to_a_beeper) {
  pick_beeper
  move
}
```

(a) How many beepers will BB pick up?

(b) If the "while" is replaced by "if" how many beepers would BB pick up?

3. Caution this one contains some tricks!
World;

```
robot 5 5 N 7
```

Program:

```
define turnsouth {
     while (not_facing_east) {
          turn_left
     }
}

define movenorthandput {
     turnsouth
     move
     create_beeper
     put_beeper
}

define main {
   do (3) {
        movenorthandput
    }
}
```

(a) Where will BB be when it turns off?

(b) How many beepers does BB drop?

(c) Will BB be next to a beeper when it turns off?

(d) How many beepers will be in beeper bag 0?

1.11.2 BeeperBot Programming Questions

1. Write a program so that when BB starts in arbitrary location (x, y) facing an arbitrary direction D the program stops with BB in location (y, x) and facing in the opposite direction from D. For example, if BB starts in $(3, 50)$ facing North then it should move to $(50, 3)$ facing South.

2. Write a program to have BB make a 5x3 rectangle outline with beepers.
3. BB starts in cell 5 5 facing north and there are an unknown number N of beepers in cell 5 5 (i.e. N beepers next to BB).
   (a) Write a program to have BB form a line of N beepers running to the north, starting from the cell BB is in. (one beeper per cell)
   (b) Write a program to have BB place a diagonal line of N beepers running to the northeast. (one beeper per cell).
4. BB starts in cell 5 5 facing north and there are an unknown number N of beepers in cell 5 5 and either 0 or 1 beepers in beeper bag 0. Write a program to have BB form a line depending on the number of beepers in her bag. If she has 0 beepers in her bag, BB should form a line of N beepers running north. If she has 1 beeper in her bag, she should form a diagonal line running to the northeast.
5. BB starts in some cell facing east. She has K beepers in her bag, for some integer K >= 0. Program BB to lay a line of beepers to the east, one per cell, until either she runs out of beepers, or she comes to a wall.
6. BB starts in an arbitrary cell, facing in an unknown direction. Have BB move to position (5,5) facing north.

1.11.3 * Advanced Questions

These questions are for those seeking more challenge or insight than the usual 101 level course.

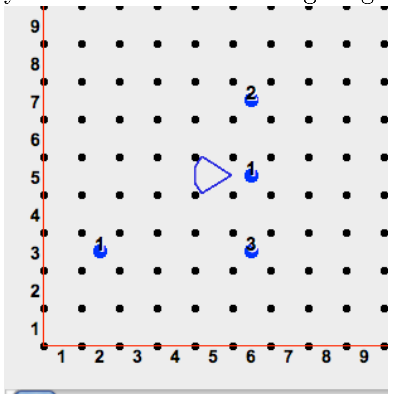
1. Given this world:

```
robot 5 5 E
beepers 6 5 1
beepers 6 7 2
beepers 6 3 3
beepers 2 3 1
```

   and this program

```
define main {
     while (front_is_clear) {
        move
        while (next_to_a_beeper) {
            pick_beeper
            turn_left
        }
    }
}
```

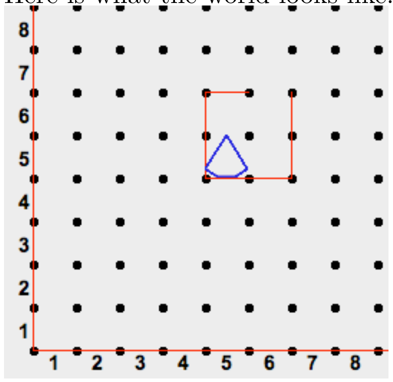
Trace the path followed by BeeperBot, indicating where she will stop. Do your trace on the following diagram.



2. Using the program from previous question, suppose you have the following world

```
robot 5 5 N
wall 5 6 N
wall 6 6 E
wall 6 5 E
wall 4 5 E
wall 4 6 E
wall 5 4 N
wall 6 4 N
```

Here is what the world looks like:

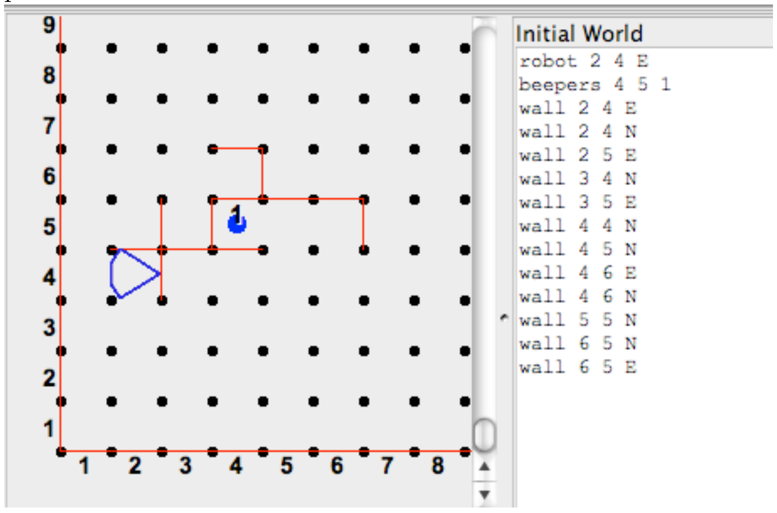


Add **beepers** statements to this world so that when the program terminates, the robot is on cell 2 1. Do not change any of the world statements already in the world, and do not change the program.

To think about: Recall that the beepers you add to the world file are input to the program. In this case, you can also think of placing the beepers as a kind of program itself guiding the robot. Now consider that BeeperBot itself is in fact a program running on your computer, that takes as input your BeeperBot code and your world file. This kind of viewing a program as data input to another program has many levels in a modern computer system.

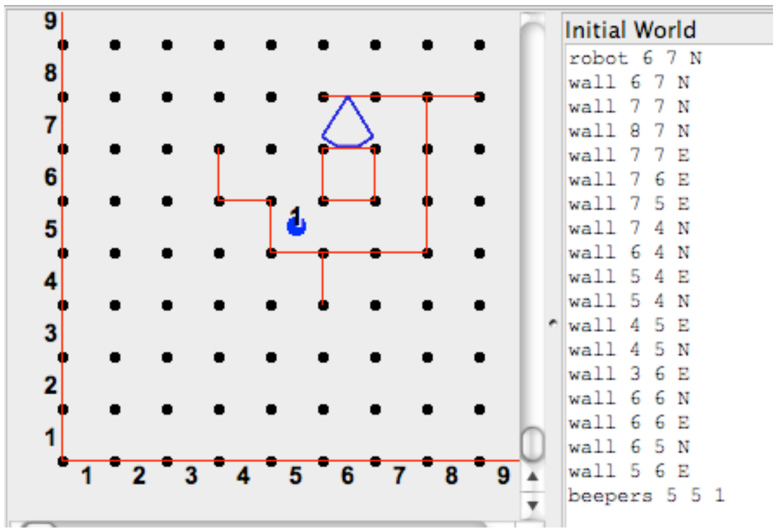
3. One way of negotiating certain types of maze is to alway keep your right hand on the wall. If when you enter a maze you move so that your right hand never loses contact with the wall, you are guaranteed that you will eventually return to your starting point. Of course, you could use your left hand instead, but don't try switching half way through!

   The goal of this exercise is to implement a rule in BeeperBot to follow a wall until she reaches a beeper. Here are some sample worlds and their pictures.

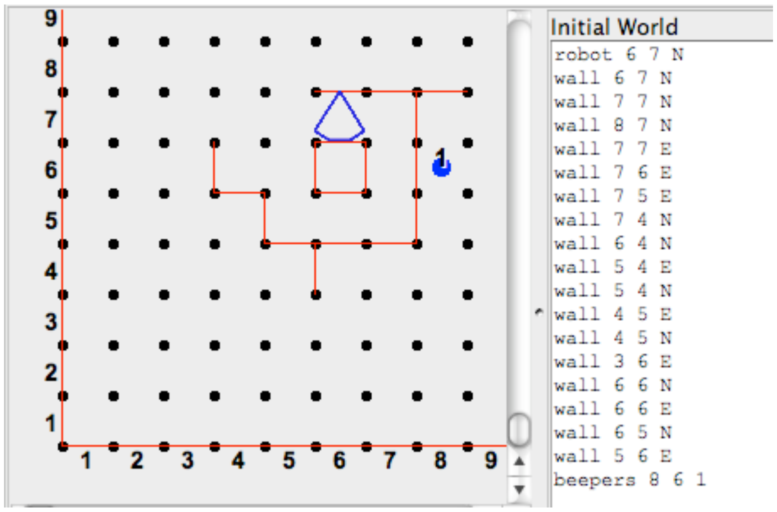


Initial World

```
robot 2 4 E
beepers 4 5 1
wall 2 4 E
wall 2 4 N
wall 2 5 E
wall 3 4 N
wall 3 5 E
wall 4 4 N
wall 4 5 N
wall 4 6 E
wall 4 6 N
wall 5 5 N
wall 6 5 N
wall 6 5 E
```



```
robot 6 7 N
wall 6 7 N
wall 7 7 N
wall 8 7 N
wall 7 7 E
wall 7 6 E
wall 7 5 E
wall 7 4 N
wall 6 4 N
wall 5 4 E
wall 5 4 N
wall 4 5 E
wall 4 5 N
wall 3 6 E
wall 6 6 N
wall 6 6 E
wall 6 5 N
wall 5 6 E
beepers 5 5 1
```



```
robot 6 7 N
wall 6 7 N
wall 7 7 N
wall 8 7 N
wall 7 7 E
wall 7 6 E
wall 7 5 E
wall 7 4 N
wall 6 4 N
wall 5 4 E
wall 5 4 N
wall 4 5 E
wall 4 5 N
wall 3 6 E
wall 6 6 N
wall 6 6 E
wall 6 5 N
wall 5 6 E
beepers 8 6 1
```

You should write one program that works on all of these and similar worlds. Remember, in each case BB should follow the wall she is initially facing until she reaches the beeper. It does not matter which direction she chooses to follow the wall.

Hint: the program solution I have in mind has a mere 11 lines of code.

4. In Section 1.10.2 we said “ It is usually possible to rewrite a BB program containing subroutines so that is has no subroutines.” We then used this idea to develop detailed state diagrams that work when there are subroutines. However, this does not always work this well. In particular, subroutines in BeeperBot can be *recursive*, an idea we have not otherwise pursued in this chapter. Consider the following code

```
define main {
  destruction
}

define destruction {
if (next_to_a_beeper) {
       destroy_beeper
       destruction
    }
}
```

starting with this world

```
robot 3 3 N
beepers 3 3 15
```

The subroutine *destruction* is *recursive* because it sometimes calls itself. Recall that a state diagram is supposed to *finite* and depend only on the source program. Explain what goes wrong with the "dot" notation when applied to this program. Hint: run the program at slow speed and watch the **Call stack** pane. Try again after changing '15' to '50' in the **Initial World**.

5. Since we mentioned *recursive* subroutines in the previous question, you may wish to explore recursive programming a bit further. Trace the following program by hand on the world below, and predict where the robot will be when it stops, and how many beepers will be in each of Bag(1), Bag(2) and Bag(3).

```
define main {
  pickup
}

define pickup {
if (next_to_a_beeper) {
       pick_beeper
       move_beeper(0,1)
       pickup
    }
    move
    if (next_to_a_beeper) {
       pick_beeper
       move_beeper(0,2)
       pickup
    }
    create_beeper
    pick_beeper
    move_beeper(0,3)
}
```

and the world

```
robot 3 3 E
beepers 4 3 2
beepers 5 3 2
beepers 7 3 1
```

Now run the program and check your answers.

1.12 BeeperBot History

BeeperBot is a program written as a CMPUT 401 course project at the University of Alberta in the winter of 2008. Dr. Ken Wong was the course instructor, and the team members were Matthew Johnson, Timothy Lam, Mark Nicoll, Adrienne Paton, and Matthew Whitton. Joe Culberson was the client, generating the project proposal as an expansion of the basic concepts of similar beeper manipulating robots such as Karel the Robot, Guido van Robot (GvR), and RUR-PLE. So, BeeperBot is Yet Another Beeper Robot, but we did not like YABR as an acronym.

BeeperBot has three operating modes: **primitive, standard** and **auxiliary**. As the names suggest, these are related to the standard operation of GvR, Karel etc. with standard mode being the most similar.

Even in **standard** mode, there are some differences in the syntax of the programs, and BB has two additional commands, *create_beeper* and *destroy_beeper* which operate on the set of beepers in the room in which BeeperBot is standing. These make it easier to emulate a Turing-equivalent model of computation. The syntax of BeeperBot uses braces for code blocks, and parentheses to delineate conditionals. This may make the transition to languages such as Java or C/C++ easier for students.

In **primitive** mode, BeeperBot has no beeper bag, and all commands related to the beeper bag(s) are disabled. In addition, the do command is disabled. It is not too hard to see that BeeperBot **primitive** is nevertheless still computationally complete. Code for this mode is also (except for recursion) easily represented within a simple class of structured finite state machines. Thus, this should be ideal for teaching concepts of state, both within code, and the state (configuration) of the world.

Auxiliary mode gives BeeperBot a total of 10 beeper bags, plus operators to move beepers between bags, and to test whether any bag is empty. All transfers to/from the world must go through bag 0, the default bag available in standard mode. Together with the in-code controls on program execution speed available in all modes, the hope is to illustrate concepts such as abstraction, memory organization, variables and other algorithmic concepts as typically implemented on a RAM/RASP (von Neumann architecture), without the necessity of introducing a high level language and all the attendant issues of syntax and compilation that that entails.

Chapter 2

Representation: Giving Meaning to Computation

So far, the programs with BeeperBot have been designed to show how the programming language works, some concepts such as code blocks, conditional statements and loops, and to show certain concepts of computing, such as state and abstraction. However, the programs have been largely meaningless outside of BeeperBot's world. This initial meaninglessness is deliberate; it must be kept in mind that the core computational model never has any understanding. BeeperBot is a slave to the code that is written, and follows instructions blindly. BeeperBot has no understanding of the programmer's intentions.

But to be useful to us we must somehow attach meaning to the computation. That is, we must *represent* aspects of the world that are of interest in such a way that the computation reflects those aspects accurately.

Representation is deeply entwined with the design of effective algorithms, and thus all aspects of computing. In this part of the course, we will consider some aspects of representation, still using BeeperBot as our computational model.

To summarize we present a simplified list of steps for learning to program.

1. Learning the language: For BeeperBot this should only take a short time because the language is very elementary. For modern general purpose languages like C++ this can require major effort.

2. Learning how to use the language to do elementary "mechanical" tasks: In Chapter 1 we showed how to move piles of beepers and the exercises and labs led you to develop algorithms to follow paths of beepers, or sweep all beepers from an enclosed space.

3. Learning how to represent and model real world problems: In the next session we discuss representing and displaying text, and in the second section we discuss the more difficult task of representing numbers and doing elementary arithmetic.

2.1 Characters and Strings

All knowledge in our world can be communicated by strings of characters. We start with an alphabet, say a. . . z, and string them together into words, such as "words", which are separated by spaces. But the spaces themselves may be thought of as characters, called blanks. These sequences of words are broken into lines for convenience, but in computers these line breaks are special characters that indicate when the computer should start to display a new line. Sets of lines may be broken up into pages, and sets of pages are broken up into books, and sets of books form libraries. But in the end, we can if we choose think of all the writing of all of mankind as a huge string of characters.

Pictures also may be thought of as strings of characters. The typical computer or television screen is made of thousands or millions of tiny pixels which are turned off or on, tuned as to brightness, and color. The information for a pixel is stored as a sequence of *bits.* Bits are characters drawn from a binary alphabet, that is an alphabet of size two, typically represented by '0' and '1'. The entire photo you display on your background is a huge string of these symbols in the computer.

But it is not just text and photos that can be thought of as being strings of characters. Your DNA is typically viewed as a string of characters, encoded as certain chemicals. These strings are instructions that are read and interpreted by complex biological mechanisms to construct you.

In this section we show how we can represent a simple alphabet, A. . . Z, in two ways using BeeperBot. Why two ways? The best representation for a machine is not typically the best representation for a human.

We consider one representation that is easy to interpret and manipulate with BeeperBot code, and is also easy to input through a sequence of beeper piles in the **Initial World** description. We have BB convert this sequence into a display that, if you kind of squint a bit, looks roughly like the letters you would expect to see. We call this display the *output.*

Here are the two representations we use.

Input On the bottom row, starting at the position (1,1), we place piles of beepers with the following representation: $A \Leftrightarrow 1, B \Leftrightarrow 2, \ldots Z \Leftrightarrow 26$.

Output Display the characters by placing beepers to make crude letters as we typically see them.

Actually, we only make our program deal with 'A' and 'B'. We call this baby version program BABA, since it is about the only word it can display.

The representation in the sequence in the bottom row is the *input* to BABA. Here is something to consider. The **Initial World** is input to BeeperBot, and when BeeperBot is **Reset**, this creates the sequence of beepers in the World which is then taken as the input to BABA. One might say BeeperBot feeds BABA.

In addition to the two character representations, BABA also uses the second row as **memory storage.** On this row, it keeps a marker (i.e. a beeper) to remember which column the next output letter should start in.

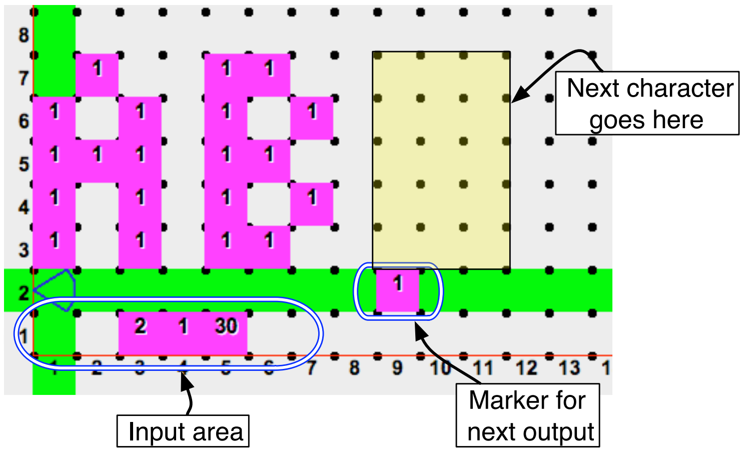


Figure 2.1: BABA

```
main
    go to the home position
    place a marker on the 2nd row, first column
    translate

translate
    go to the home position
    move east along the first row to find the next input
    count and destroy to determine the character
    if character is end-of-input
        turn_off
    move to the west end of the second row.
    move east until the output marker is found.
    destroy the output marker
    lay out the pattern of the next letter
    create a new output marker
    repeat translate
```

Figure 2.2: Pseudo-code Description of the BABA Program

In Figure 2.1 we illustrate these by looking at a screen capture of a partial computation. Note the input has a pile of 30 beepers at the end. This pile is big enough that it cannot be mistaken for a character and is used to indicate that there are no more characters in the input. Such a symbol is called an end-of-input *eoi* or end-of-file *eof* character. See question 1 in the sample questions at the end of this chapter.

Given this, we can describe the program that does the translation at a high level using pseudo-code. *Pseudo-code* is a kind of *abstraction* where we present the overall outline of a program in statements that are concise, but which do not execute on any computation machine. The description is found in Figure 2.2.

If you look at the code of BABA, you will find that a large part of it is simply code to produce the pattern for the two letters. Such a set of descriptions for a complete alphabet is called a *font.* The document you are now reading is represented in your computer by a sequence of characters and a set of fonts saying how to display them. Here are some example font variations **bold**, *italics*, large, tiny and of course we can also change color to red or green. You will likely agree the font we provide in BABA is fairly crude in comparison.

Another consideration in choosing the machine representation is **memory or resource usage**. Consider that if we want ten 'A's we only need ten beepers on ten cells using our input format. But we would need 5 rows by 40 columns and 100 beepers to store these same characters in output format. Similarly, if a modern computer had to store all the font information for each character in a word document, then it would require a huge amount of memory for even a short document. Instead, it stores the font information for each character once, and uses a short representation for the characters themselves (Typically using ASCII which we will briefly discuss later.).

For study purposes, you do not need to memorize the BABA code. The intent of this section is merely to point out some algorithmic ideas, and give the first clue that representation is important to the ease of designing algorithms.

2.2 Questions on Text Representation

The BABA questions here should be seen as mostly training for general programming, and students should not waste time memorizing BABA code in any detail. The questions after **"To Think about"** may be relevant to later parts of the course.

2.2.1 BABA and Text Questions

Note: Although the BABA program in its current form will run under primitive mode, for these questions you will find it easier to use BeeperBot in Auxiliary mode.

1. What would happen if the BABA program were to run on an initial world that did not have an end-of-input marker?
2. Currently the BABA BeeperBot program in the sample programs only recognizes and places letters A and B. Modify the program so that it can also translate 3 to 'C', and 4 to 'D'. Add more letters if you wish.
3. Currently BABA outputs the letters with their bottom pixel on the third row.
   (a) Modify the BABA program so that initially part of the input will specify which row to put the output letters on. Note that all letters should display on the same row so you only need to specify one such number. You may assume that it will always be at least row 3, or greater.
   (b) Add another input to specify which column the first letter should be placed in. In this way, you can place the output any where on the screen (above row three).
4. Suppose we want uppercase letters 'M', 'W' and 'I'. The letters 'M' and 'W' are wider than the other letters so far, and 'I' may be thinner. If these letters are added to our font, will the BABA program handle these correctly? Try it, and explain this generality of BABA. (Fonts in which different characters have different widths are called variable width fonts.)
5. Assume BeeperBot is using **auxiliary** mode. Modify the program so that instead of using a marker on the second row to remember where the next beeper goes, it keeps track of which column the next character goes in by storing beepers in a beeper bag.

   To Think About

6. Suppose we want to represent upper and lower case letters, as well as digits, punctuation etc. Is there a better input representation than the one we have chosen? In what way could it be better?

7. ** Reverse Translation.

   (a) Suppose you have a character in its output representation, and BeeperBot is located on the bottom left cell of the 3x5 display area. Write a new program to put the numeric value of the letter in the home location (1 1). Thus, if the letter is A, it should put 1 beeper in location (1 1). If a B then put 2 beepers.

   (b) How would you extend this if you had 4 or more letters in your set of representations? (see question 1).

   (c) Now extend this to convert a string of characters to the numeric (internal) representation.

8. ** Continuing the issues of the previous question, suppose BeeperBot starts on a cell in a letter, but you do not know which cell. How would you write a program to recognize which letter BB is on?

Questions 7 and 8 indicate the difficulty of recognizing characters from images using a computer. On the other hand people much prefer images — think how difficult reading would be if you really did have characters represented as numbers from 1 to 26, or 1 to 52 if you want upper and lower case.

This difficulty is one of the problems with trying to get machines to read hand writing, and thus the continued necessity of using keyboards. On the other hand, the distinct ability of humans to recognize even severely distorted characters and the difficulty of writing programs to do the same, is now used in a clever way as both an anti-bot device to protect websites from certain types of programs, and at the same time to assist the digitization of old books. Read more about reCAPTCHA at this site `http://recaptcha.net/`.

2.3 Counting and Number Systems

Consider this sequence of characters **“123”**. You probably think that this is a number, which is also represented by the character sequence **“one hundred and two tens and three’**. You might also agree that the following also represents the same number **“three and twenty and one hundred’**. Note that aside from replacing “two tens” with “twenty” we also reordered some of the words. Now consider **“321”**. You will likely agree that this is not the same number, but why exactly?

The number system we use in our everyday lives is a positional base ten system, with the digits ‘0’. . . ‘9’. There are many other systems that have been used throughout history. To explore the variety of systems use Google to search for “number systems”, “Roman Numerals” and “babylonian number system” as a starting point.

In addition to the base ten system, we will also look at the binary number system in this course, as it is the arithmetic system used in digital computers.

But first we will take you all the way back to elementary school and the idea of numbers and counting using BeeperBot.

2.3.1 Unary Arithmetic

Suppose that you were born before the invention of writing and number systems. You have a flock of sheep to tend. At night you want to be sure all the sheep enter the fold, before you retire to watch TV. (Well, okay, maybe there was no TV then either.)

How can you be sure that some sheep has not gone astray? Well, what you have is a bag full of pebbles, one pebble for each sheep. You dump the pebbles on a board, and as each sheep enters the fold, you drop a pebble back in the bag. If, when the last sheep enters, all the pebbles are back in the bag, then you must have all the sheep. If there are pebbles left over, then you need to go find the missing sheep. And if you run out of pebbles before the last sheep enters the fold, you are likely to receive an irate phone call from a neighbor. (What, no phones?)

These pebbles are a **unary representation** of the number of sheep you have. This matching of pebbles to sheep is in fact precisely what counting is. Suppose you wanted to compare the size of your flock to that of your neighbor. Since sheep run around it might be hard to tell which of two large flocks is the larger. But if you compare your bag of pebbles to your neighbors the task is much easier.

You simply match the pebbles up in pairs, one from each bag, and the bag that has some left over represents the larger flock. In fact, you also have the difference as the number of pebbles that are left over.

If you want the total of your two flocks, simply dump the two bags together.

In the sample programs, we will show BeeperBot programs to do arithmetic using unary representation. Example programs for addition, subtraction and multiplication will be discussed.

BB Unary Addition

We will assume BB is facing two piles of beepers, as shown in figure 2.3 and wants the total placed in the third square. One of the requirements we make is that the inputs are preserved.

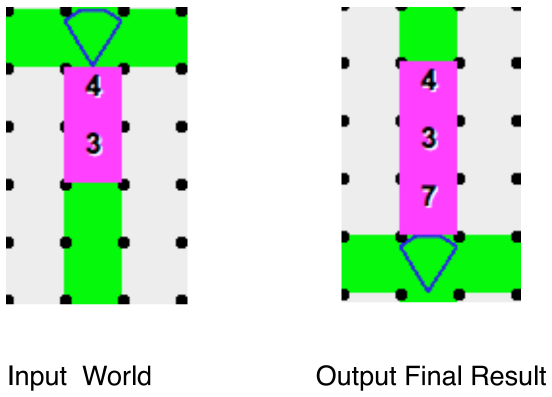


Figure 2.3: Unary Addition

Here is one sample code that will produce this. First the main program.

```
# unary addition example, not destroying inputs
define main {
    # get the first input
    move
    get_and_restore

    # get the second input
    move
    get_and_restore

    # output the result
    move
    display_sum
    move
}
```

Next is the **get_and_restore** subroutine. Recall that pick_beeper always puts the beeper in bag 0. However, we need to restore the pile of beepers, so we do this by moving each beeper to bag 1, and each time one is moved, we also create one beeper. Notice this matching is exactly how we described counting sheep above. We also include the display_sum routine, which merely moves the beepers to bag 0 then drops them.

```
# pick up the beepers and put them in
# bag 1, at the same time recreating
# the pile
define get_and_restore {
    while (next_to_a_beeper) {
        pick_beeper
    }
    while (has_beeper(0)) {
        move_beeper(0,1)
        create_beeper
    }
}

# drop all the beepers from bag 1
# RECALL put_beeper only works on
# bag 0
define display_sum {
    while (has_beeper(1)) {
        move_beeper(1,0)
        put_beeper
    }
}
```

Unary Subtraction

Subtraction presents us with new difficulties. Suppose we subtract $4 - 10$. The answer is negative, namely -6. But BB cannot print a minus sign, so we have to choose some other representation. We will adopt the notation that putting one beeper to the left of the result means it is negative. This is illustrated in figure 2.4.

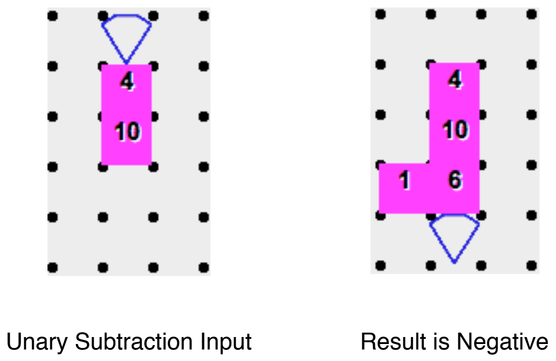


Figure 2.4: Unary Subtraction

The code used in this subtraction program is found in figure 2.5, except for turn functions defined previously.

```
# unary subtraction example
define main {
    # get the first input
    move
    get_and_restore
    # we have to keep
    # the two inputs separate
    move_1_to_2

    # get the second input
    move
    get_and_restore

    # output the result
    move
    display_difference
    turn_south
    move
}

# pick up the beepers and put them in
# bag 1, also recreating pile 1
define get_and_restore {
    while (next_to_a_beeper) {
        pick_beeper
    }
    while (has_beeper(0)) {
        move_beeper(0,1)
        create_beeper
    }
}

define move_1_to_2 {
    while (has_beeper(1)) {
        move_beeper(1,2)
    }
}
```

```
# recall the fist value is in bag 2
# and the second is in bag 1
define display_difference {
    while (has_beeper(2)) {
        if (has_beeper(1)) {
            # this part does matching
            # bag 9 is used for trash
            move_beeper(1,9)
            move_beeper(2,9)
        } else {
            # if bag 2 had more, result is
            # positive so put it down
            move_beeper(2,0)
            put_beeper
        }
    }
    # if bag 2 became empty before bag 1
    # we have a negative result
    if (has_beeper(1)) {
        turn_west
        move
        # this beeper is the negative sign
        create_beeper
        turn_east
        move
    }
    # put down beepers in bag 1
    # note: if result is postiive
    # this loop does nothing.
    while (has_beeper(1)) {
        move_beeper(1,0)
        put_beeper
    }
}
```

Figure 2.5: Subtraction Code

On the left side is the main code. This is similar to addition, except the inputs are kept separate, the first going to bag 2, before the second is put in bag 1. On the right side the display difference is somewhat more complex. In the first while loop, as long as both bags have beepers, we simply trash one from each (by moving to bag 9). If bag 1 runs out before bag 2, then the remaining ones in bag 2 are moved to the pile. This is the case when the result is positive (or zero). On the other hand, if bag 2 runs out first, then the loop exits with some beepers remaining in bag 1. This indicates to BB that it should put down a minus sign, which occurs in the ‘if’ following the first while. Then BB puts the remainder of bag 1 on the pile.

Note the second ‘if’ and the second ‘while’ do nothing if bag 1 is empty, which is the case when the result is not negative.

Unary Multiplication

When we multiply two numbers, say, 2 by 3, we actually mean add 2 three times, i.e. $2+2+2$. We can use this simple idea to teach BB to do multiplication. In unary representation, this should be quite simple because it basically requires BB to pile up beepers. If BB can create multiple copies of the pile, then the rest is very simple. So, given a pile of beepers, how do we teach BB to copy this pile of beepers into another pile? BB can use one of the bags to store the original pile of beepers, and then another one to store the newly created pile.

The code is displayed below. The only significant difference between this and the code in unary addition is that it copies the second pile n times, where n is the number that was in the first pile. Like the subtraction, it must keep the two piles in different bags, so it puts the first pile in bag 2 after getting it. Then it moves these to another bag one by one, adding a copy of the second pile for each beeper moved.

```
# unary multiplication example,
# with two inputs
# not destroying inputs
define main {
    # get the first input
    move
    get_and_restore

    # move the first input to  bag 2
    while (has_beeper(1) ) {
        move_beeper(1,2)
    }

    # go to the second input
    move

    # for each beeper in bag 2,
    # add another copy of input 2
    # -- This illustrates why not destroying
    # -- inputs may be important!
    while (has_beeper(2)) {
        move_beeper(2,3)
        get_and_restore
    }
    # output the result
    move
    display_product
    move
}

define get_and_restore {
    while (next_to_a_beeper) {
        pick_beeper
    }
    while (has_beeper(0)) {
        move_beeper(0,1)
        create_beeper
    }
}

define display_product {
    while (has_beeper(1)) {
        move_beeper(1,0)
        put_beeper
    }
}
```

2.3.2 Counting Questions

Review the unary subtraction, as it will not be covered in class this term.

1. Write a unary addition program that allows for signed arithmetic. Use the same sign convention as the unary subtraction program, namely that there is a single beeper to the west of each input if the number is negative.
2. Write a program that will add a column of values in unary. The first location should be the number of numbers to be added. If this is n, the following n piles of beepers are the numbers to be added.
3. ** How would you do division in unary? If you divide 31 by 3, what does it mean? Consider both quotient and remainder.

2.4 Base Ten Arithmetic

According to the web site *'Agriculture - Sheep/Wool Industry - Australia'*[1] Australia had approximately 102 million sheep in the 1990s. If we tried to represent that number with a bag of pebbles, it would require a very large bag. Note that the string of characters "120 million" or the equivalent string "120,000,000" is much smaller and easier to manipulate. Furthermore, adding say 60 million to 50 million would require the BeeperBot unary program to execute hundreds of millions of program steps, picking up beepers, creating beepers and putting down beepers. Yet, most likely you have already realized the total is 110 million, and you certainly did not take 110 million steps to discover this. Unary computation is not efficient.

As noted earlier, the choice of representation can make an enormous difference in how hard it is to do a computation. Also, as noted in the introduction, different societies throughout history developed different representations. If you are interested in a bit of frustration, consider the difficulty of writing a program to add two numbers using the Roman numeral[2] notation. Of course there were also other number bases used such as the Babylonian System[3] which used base 60. If you thought memorizing the 10 times table in grade school was onerous, imagine learning the 60 times table!

Likely the base ten system has come to dominate in the world of human arithmetic in large part because it corresponds to the number of fingers most of us have, and for the range of numbers used in every day transactions, it provides a good balance between the number of digits used to represent a number and the number of different symbols that need to be remembered. To be fair, the Babylonians did not require 60 different symbols, but rather compositions of two symbols[4]. Twenty was also sometimes used as a base, apparently because unshod people could also use their toes. And occasionally 5 was used.

In this section we are going to take you all the way back to elementary school to recall the algorithms for doing arithmetic using the base ten system. You will then be asked in Lab 3 to implement a simple base ten addition algorithm for BeeperBot. Our expectation is that once you have done this it will be easier for you to understand the binary (base 2) number system, by seeing the parallels to the base ten system. Variations of binary number representations are used in all modern computers to do arithmetic. In addition, we want to reinforce the concept that our representation of numbers is inherently intertwined with algorithms. In fact, we could argue that most scientific knowledge is encoded as an interaction between representation and algorithms.

[1] `http://www.anra.gov.au/topics/agriculture/sheep-wool/index.html#howmany`

[2] `http://en.wikipedia.org/wiki/Roman_numerals`

[3] `http://www.math.wichita.edu/history/topics/num-sys.html#babylonian`

[4] `http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Babylonian_numerals.html`

2.4.1 Counting in Base Ten

In Section 2.3.1 on Unary arithmetic, we said the fundamental concept of counting was matching, in the sheep case matching pebbles to sheep. Recall that the bag of pebbles indicated the number of sheep provided we could match each pebble to one sheep.

Is there a corresponding notion in base ten? Yes, there is. Using the symbols '0'. . . '9' and some simple rules (i.e. algorithm) we can generate an unbounded sequence of strings of digits. The first string we start with is '0', and as you know this means no sheep has passed the gate. When a sheep goes by the last digit is increased by using the rules $0 \to 1$, $1 \to 2$ and so on, where $\to$ can be read as "becomes". The only tricky part is when the rightmost digit is a '9'. In this case the '9' is replaced by '0', and the rule is applied to the next digit, and so on.

Here is an example of counting, with 's' standing for 'sheep'.

0	
1	s
2	ss
. . .	
9	sssssssss
10	ssssssssss
11	sssssssssss

There are two ways to interpret this list. The *matching, or counting*, view is to match the string of digits to the last sheep in the corresponding list. The *cardinality* view is to use the string to identify the total quantity of sheep in the corresponding line. Notice how much easier it is to interpret '11' than the corresponding list of 's's. Is it in fact correct?

This should already be familiar to you, and so we leave the rest for you to think about. In the exercises at the end of this section you are asked to program BeeperBot to count in unary and in base ten.

2.4.2 Conversion of Unary to Base Ten

Suppose I have a large collection of pennies. How do I get the base ten representation? Well, consider using dimes and loonies. The first step is to break the group into groups of ten like this.

```
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppppppppp
pppp
```

Notice that there are 4 left over. Now each set of ten pennies can be replaced by a dime.

```
dddddddddddd
pppp
```

Now, we create groups of ten on the dimes.

```
dddddddddd
dd
pppp
```

and we replace each (there is only one in this example) group of dimes by a loonie.

```
L
dd
pppp
```

We now see that we have 1 loonie, 2 dimes and 4 pennies, so we started with 124 pennies.

Wait! There was some magic there. Our base ten monetary system uses a *loonie* to represent ten *dimes* and each dime represents ten *pennies.* We could extend this with the $10, $100 and $1000 bills, to get higher *powers of ten*, representing 1000, 10000 and 100000 pennies respectively. We say that 100000 is the 5th power of 10, written in shorthand as 10^5, because it is the result of multiplying 10 by itself 5 times.

Our base ten number system uses *position* to represent *powers of ten.* Thus, 124 means $1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$. Note that we define 10^0 to be 1, and frequently we do not write 10^0 because multiplication by one has no effect. But we write it here for completeness.

Again, note the efficiencies introduced by using base ten. We have replaced 124 pennies by 1 Loonie, 2 dimes and 4 pennies, or a total of 7 coins.

2.4.3 Addition Base Ten

Suppose we want to add two numbers such as $685 + 417$. We will think in terms of money and position.

So, 685 is 6 Loonies, 8 dimes and 5 pennies and 417 is 4 Loonies, 1 dime and 7 pennies.

The total is 10 Loonies, 9 dimes and 12 pennies. In positional form

6	8	5
+4	1	7
=10	9	12

Of course we don't usually write our sums that way. So we remove a group of 10 pennies and replace it with 1 dime, positionally speaking.

10	9	12
=10	10	2

Then we replace the 10 dimes with 1 Loonie

10	9	12
=10	10	2
=11	0	2

and finally we replace ten Loonies with one 10-dollar bill.

	10	9	12
=	10	10	2
=	11	0	2
= 1	1	0	2

It is probably worth noting that we simply applied our algorithm from the previous subsection on each column, after adding in the carry. You have probably been taught in grade school to do the carry as you add each column. To think about: see that these two approaches produce the same result.

2.4.4 Programming BeeperBot to Work in Base Ten

In Lab 3 you will be asked to complete a BeeperBot program to do addition in base ten. Basically the algorithm you use is a direct implementation of the algorithms described so far, where beepers replace pennies and columns in BB's world represent position.

The key to your task is to have BB replace a group of ten in one pile with one beeper in the next pile to the west. However, there is a difficulty. BB does not know the number ten specifically. All BB can ask is whether one or more beepers exist in a specific bag or the location where it stands.

For example, suppose you have picked up the current digit and put them in bag 1. Now you want to remove a block of ten beepers, creating one carry if there are ten to start with. You might write a loop such as this

```
do (10) {
   move_beeper(1,4)
}
```

However, if there are only 7 beepers in bag 1 when the loop first starts then there will be a run time error saying no beeper to move.

Well, you can try a fix like this

```
do (10) {
   if (has_beeper(1)) {
       move_beeper(1,4)
   }
}
```

This will eliminate the run time error, but the problem now is that when you finish the loop you do not know if BB ran out before the loop finished or not.

There are two possible fixes to this problem. One is to add an 'else' to the 'if' to create a beeper and put it in say bag 3 whenever bag 1 is empty. After the loop is finished, if bag 3 has a beeper then you know there were less than 10 beepers in bag 1 when you started.

The other way is to replace 'do(10)' with 'do(9)'. Then after the loop finishes, if there is still a beeper in bag 1 you know there were at least ten to start with.

With either approach you still have to figure out what remains as the current digit, and whether or not there is a carry. These hints should be enough for you to complete the Lab, but you should look at the code on the web site, and think about it deeply before going to the lab.

The next page has the lab stub (except the turning routines which you have seen before) for easy reference.

The first routine you have to fill in is basically the same as the one we had for unary addition.

The second routine is the more complex one. It must compute the carry and how many remain in the current digit.

```
### PART 1: STUDENTS HAVE TO FILL IN THE FOLLOWING
#Choose a bag to keep the sum in.
define add_digit_to_sum {

}
### END OF PART 1


### PART 2:  STUDENTS HAVE TO FILL IN THE FOLLOWING ROUTINE AND
### ADD ANY ADDITIONAL ROUTINES THEY NEED TO FINISH
define put_digit_and_save_carry {

}
### END OF PART 2.

# this routine takes BB to the spot above of the next column of digits.
define go_to_next_column {
      turn_around
      move
      move
      move
      turn_west
      move
      turn_south
}

#main program
#assumes robot starts just above
#the inputs, facing south.

define main {
# add three columns --
# the fourth iteration is in case of a carry out

    do (4) {
          move
          add_digit_to_sum
          move
          add_digit_to_sum
          move
          put_digit_and_save_carry
          go_to_next_column
    }

}
```

2.5 Base Two or Binary Arithmetic

Aside from the fact that the typical human has ten fingers, the choice of ten as the base of our number system is arbitrary. As we will see in later parts of the course, for modern computers a base of two is a more reasonable choice.

In a binary system there are only two digits to worry about, 0 and 1. Like the base ten system, it is *positional* with the digits representing powers of 2 instead of powers of 10.

There are many web sites that provide further reading on binary numbers'[5] which can be found by doing a google search on "binary numbers".

In our study of binary numbers, it is probably easiest to simply start counting. Counting is simply the process of repeatedly adding 1 to our current number. As in base ten, in binary we also start with '0' representing zero.

2.5.1 Counting in Binary

The binary counting process parallels the method of counting in base ten. Recall that in base ten, when we add one to a number, we start with the rightmost digit and increase it by 1. If it is '9' then we change it to a '0' and carry one to add to the next digit.

In base two, we also start by considering the rightmost digit and add one to it. If the current digit is a '0', then adding one changes this to a '1' and the digits to the left are not changed. If the current digit is a '1', then adding one changes it to a '0', and we carry one to add to the next digit to the left.

So, the following represents the first few numbers counting in base 2 and in base ten

Base 2	Base 10
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

[5]`http://en.wikipedia.org/wiki/Binary_numeral_system`

To continue counting, start at the right most digit of the number and add 1 according to the two rules. So for example, to get the next binary number after 1111 we make the rightmost digit 0 and carry 1 to add to the next. Again, this means the next digit is zero and carry one to the next. This gets repeated again, and we get a 4th zero, with a carry of 1. This time there is no one to start with and so we get 10000 as the binary representation of sixteen.

Here is an informal algorithm for incrementing a binary number by 1.

```
position is set to 0.
carry is set to 1.
while (carry equals 1) {
     if (digit at position is 1) {
         set digit in position to 0
         add one to position
     } else {
         set digit in position to 1
         set carry to 0
     }
}
```

To count sheep, initialize the string to '0' and execute this algorithm once for each sheep that passes.

2.5.2 Conversion to Binary

Suppose we wish to represent the number of sheep in 'sssssss' by a binary number. First, we group into sets of two 'ss ss ss s' noting we have one left over. We replace the pairs by single 't's to represent "twos". Thus, 'ttts'. We now break t's into groups of two 'tt t s' and replace each pair with an 'f (for four). This gives 'fts', that is there is one four, one two and one one. Writing it in the usual form we have '111'.

This generalizes, in the same way to arbitrary numbers. For example, $ssssssss \rightarrow tttt \rightarrow ff \rightarrow e$, where e is eight or 1000, since there are no f's, t's or s's left over.

Can we express this more succinctly? Yes, by using division. Suppose we start with 157 in base ten. We divide by 2 to get 78 with remainder 1. This corresponds to 1 's' left over, so we know the rightmost (0th) bit is 1. We then divide 78 by 2 to get 39 with 0 as a remainder. No remainder means there is no t. We continue this until the number becomes 0. Here is a table of the complete conversion, where each row after the first is obtained from the previous by dividing by 2. The remainder is always 0 or 1, and forms the binary number from right to left.

Number	Remainder	Binary
157	1	1
78	0	01
39	1	101
19	1	1101
9	1	11101
4	0	011101
2	0	0011101
1	1	10011101
0		

Here is an algorithm for conversion to binary.

```
while (number is not 0) {
     if (number is even) {
         add 1 to the left of the binary representation
     } else {
         add 0 to the left of the binary representation
     }
     divide number by 2, and drop any fraction
}
```

2.5.3 Conversion of Binary to Base Ten

To illustrate this, let us do an example, $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, where the numbers on the right are in base 10. Now $2^3 = 2 \times 2 \times 2 = 8$, $2^2 = 4$, $2^1 = 2$ and $2^0 = 1$. Also, 1 times anything is just 1 so we see that $1101_2 = (8+4+0+1)_{10} = 13_{10}$ as expected. Here we have added the subscript 2 to the binary and 10 to the base ten numbers just to clarify which base we are working in.

Taking the binary number from the previous section we have $10011101 = 2^7 + 2^4 + 2^3 + 2^2 + 2^0 = 128 + 16 + 8 + 4 + 1 = 157$. It is good to see that this agrees with the number we started with in the previous section.

Here is an algorithm for conversion from binary to base ten.

```
set power to 1
set result to 0
while (there are more binary digits) {
     if (next digit is 1) {
         add power to result
     }
     multiply power by 2
}
```

When this finishes, **result** will hold the result we want. Notice that when the next digit is 0, we do not need an else, since adding 0 to the result has no impact. But every time we go around the **while** loop we must double the power for the next iteration.

2.5.4 Addition in Binary

To add two numbers in binary we must first memorize the base two addition table. Here it is, it should not take you too long.

$+$	0	1
0	0	1
1	1	0 /1

The bottom right entry says that $1 + 1 = 0$ with carry 1. All other entries have carry 0.

Now suppose we want to add $110101 + 11001$. The addition would look like this, where the c's indicate there is a carry in of 1.

```
 cc  c
  110101
   11001
 -------
 1001110
```

It is always a good idea to check results when we can.
$110101_2 = (2^5 + 2^4 + 2^2 + 2^0)_{10} = (32 + 16 + 4 + 1)_{10} = 53_{10}$.
$11001_2 = (2^4 + 2^3 + 1)_{10} = (16 + 8 + 1)_{10} = 25_{10}$.
$1001110 = (2^6 + 2^3 + 2^2 + 2^1)_{10} = (64 + 8 + 4 + 2)_{10} = 78_{10}$.
$53 + 25 = 78$ and this agreement adds confidence that our result is correct.

2.5.5 Multiplication in Binary

Here is the multiplication table for binary numbers.

$\times$	0	1
0	0	0
1	0	1

Again, this should be fairly easy to memorize. Note there are no carries generated by this table.

Multiplication works as it does in base ten, except it is much easier. Here is an example.

```
    1101
     101
 -------
    1101
   0000
  1101
 -------
 1000001
```

Notice that where there is a one in the multiplier, we simply copy the multiplicand. Otherwise, we insert a row of zeroes. Just as in base ten, each row is offset by one space to the let from the row above.

Again, let us check the answer.
$1101_2 = (2^3 + 2^2 + 2^0)_{10} = (8 + 4 + 1)_{10} = 13_{10}$.

$101_2 = (2^2 + 2^1)_{10} = (4 + 1)_{10} = 5_{10}$.
$1000001_2 = (2^6 + 2^0)_{10} = (64 + 1)_{10} = 65_{10}$.
And checking we also find $13 \times 5 = 65$.

2.6 Arithmetic Questions

1. Program BeeperBot to count in Unary. On the first row put a pile with one beeper, on the second a pile of two beepers, and so on. This should continue until the user presses the **pause** or **reset** button.

2. Program BeeperBot to count in base ten. Starting near the bottom, and say in column 20, BeeperBot should represent the number one on the first row, then two on the second and so on until the user pushes the **pause** or **reset** button. You will likely need to set some kind of boundary markers, say piles of 11 beepers, one on each end of each number so that BeeperBot will know when to stop, especially if you are using no beepers to represent the number 0. If you are using some other representation of the digits, be sure to describe it.

3. Write a BeeperBot program to convert a base ten number to unary. Assume the robot starts at right end of a string of digits, represented by piles of 0 to 9 beepers, and to the left of the last digit there is a wall. Once the wall is reached, move north one step and deposit a pile of beepers equal to the number represented in base ten.

4. Here we consider a base 5 system. It may help you to think of this in monetary terms. You can use pennies, nickels and quarters, and you get a new coin called the 'cubie' which is worth 5 quarters, or 125 pennies.
   (a) In base ten we use digits '0,1,2,3,4,,6,7,8,9'. In base 5, what digits do we need?
   (b) Convert the number represented 137 in base 10 into its equivalent base 5 notation. Use the monetary idea to help if needed.
   (c) Complete the addition $134 + 242$ in base 5.

5. Practice converting numbers to and from binary. Start with a base ten number, convert it to binary, then convert the result back to base ten, and check your answer. Repeat this several times until you are confident.

6. Practice adding numbers in binary. Check your answers by converting to base ten and check your results.

7. Practice multiplying numbers in binary. Check your answers by converting to base ten and check your answers.

8. Explain **why** the binary multiplication technique in Section 2.5.5 works.

Chapter 3

Logic, Circuits and Architecture

We have now seen that the base 10 number system that we are familiar with can be implemented directly, if somewhat clumsily, using the BeeperBot model of computation. We also saw that there are other choices of number systems, in particular the binary number system.

It is time to move on now to a different computational model, the one that underlies modern computers. We may see BeeperBot again briefly when we discuss algorithms, but otherwise we will leave our little robot behind now. However, the basic coding constructs, such as **while** and **if** that we have learned will be used again later when we use the Program Exploration Tool, which is a tool for understanding the underlying mechanisms of a language that is similar to modern languages such as C. You will likely also notice other concepts that carry forward. We certainly intended that you should.

But for now, we switch our focus to the world of false and true, or 0 and 1.

3.1 Binary Number System Variations

We introduced binary numbers at the end of the last chapter as an alternative to the familiar base ten system. Recall that binary numbers are called binary because they use only two digits, 0 and 1. As we hinted in the introduction in this chapter, and as will be expanded on in the following sections, modern computers are built on electronic devices which have only two states, which we label 0 and 1 (or false and true) for easy reference and interpretation.

So, for continuity, we continue to develop binary number representations in this section.

3.1.1 Fractions in Binary

If we consider the base ten number 83.76 you know that it means $83\frac{76}{100}$. In fact, you can also expand this as

$$\begin{aligned}
83.76 &= 8 \times 10^1 + 3 \times 10^0 + 7 \times 10^{-1} + 6 \times 10^{-2} \\
&= 80 + 3 + \frac{7}{10} + \frac{6}{100} \\
&= 83 + \frac{76}{100}
\end{aligned}$$

Recall that a negative exponent on 10 just means divide the number by 10 that number of times. In the last line we simply combine terms over a common denominator.

The "**.**" in 83.76 is known as the *decimal* point. We also know that not all fractions have a finite decimal representation, for example $\frac{1}{3}$ has the infinite repeating representation $0.333\ldots$.

We can also expand the binary number system in a similar way. The "**.**" should now be called by the more general name *radix point*, since decimal relates to the value ten.

Suppose we have the binary fractional number 101.101. We can expand this pedantically in binary as

$$\begin{aligned}
101.101 &= 1 \times 10^{10} + 0 \times 10^1 + 1 \times 10^0 + 1 \times 10^{-1} + 0 \times 10^{-10} + 1 \times 10^{-11} \\
&= 100 + 1 + \frac{1}{10} + \frac{1}{1000} \\
&= 101 + \frac{101}{1000}
\end{aligned}$$

Note, that in the above we expressed every number, including the exponents and divisors in binary, just to emphasize our ornery, err I mean binary nature.

You are probably asking "But what is that number?" What you actually mean of course is "what is the equivalent number in base ten?". Well, we can convert 101 and 1000 using the techniques we already learned, namely (using subscripts to indicate base)

$$\begin{aligned}
101_2 &= (4+1)_{10} = 5_{10} \\
1000_2 &= 8_{10}
\end{aligned}$$

Combining these we see

$$\begin{aligned}
101.101_2 &= \left(5 + \frac{5}{8}\right)_{10} \\
&= 5.625_{10}
\end{aligned}$$

More directly, we can convert binary to decimal by noticing that the digits to the right of the binary radix point correspond to the base ten values $1/2 =$

0.5, $1/4 = 0.25$, $1/8 = 0.125$ and so on. So 0.101 in binary corresponds to $0.5 + 0.125 = 0.625$.

What about conversions the other way? Well, just as 1/3 has no finite exact decimal expansion, similarly many fractional values have no exact binary expansion. One example is that in base ten $0.3 = \frac{3}{10}$. However, this value requires an infinite repeating digit expression in binary. Since we do not require long division in binary for this course, we will leave this topic at this point, though the more adventuresome may wish to try working it out as an exercise.

We can add binary fractions exactly as we add binary numbers without fractions just as we do in base ten. Just remember to line up the radix points of the two numbers. Thus, to add 101.101 and 11.1 we pad a couple of zeroes to the second number and add like this

$$\begin{array}{r} 101.101 \\ 11.100 \\ \hline 1001.001 \end{array}$$

It is always a good idea to check your addition by converting all the numbers to base ten, and we leave that as an exercise.

We can also multiply by fractions. To multiply 1001 by 0.01 we simply shift the radix point two places, just like base ten, namely the result is 10.01. If the multiplier has more digits, we simply extend the methods we already developed, remembering where the radix point should be.

3.1.2 Finite Binary Representations and Powers of 2

Digital computers have the restriction imposed by engineering constraints that we only get a fixed number of binary digits, or *bits* as they are commonly know, to represent a number. If we want to represent numbers that require more bits then we have to use software to combine these smaller chunks. The term *word* is used somewhat ambiguously to refer to the number of bits available as single entity on a certain machine. This varies from machine to machine.

Early home computers had a limit of eight bits called a *byte.* Memory and disk sizes are still typically listed in terms of the number of bytes they can hold. A half byte, or 4 bits, is called a *nibble.* For our purposes we will use qualified forms such as a 16-bit-word, 32-bit-word or 64-bit-word. These are the most common sizes used, although other sizes have been used, such as a 36-bit-word as well as other sizes.

In figure 3.1 we illustrate the effect of having a word size of 4-bits. The first thing to note is that if we have n bits, then we have 2^n different bit patterns that we can represent. Figure 3.1 displays the $2^4 = 16$ patterns when $n = 4$. Each time we add another bit, we double the number of patterns, since we have all the old patterns with 0 in the new position, plus all the old patterns with 1 in the new position.

In figure 3.1 we have listed the patterns so that traversing clockwise gives us the binary counting order, with the binary representation of zero at the top. So what is that ”Over Flow Boundary”line indicating? Well, suppose we add 1

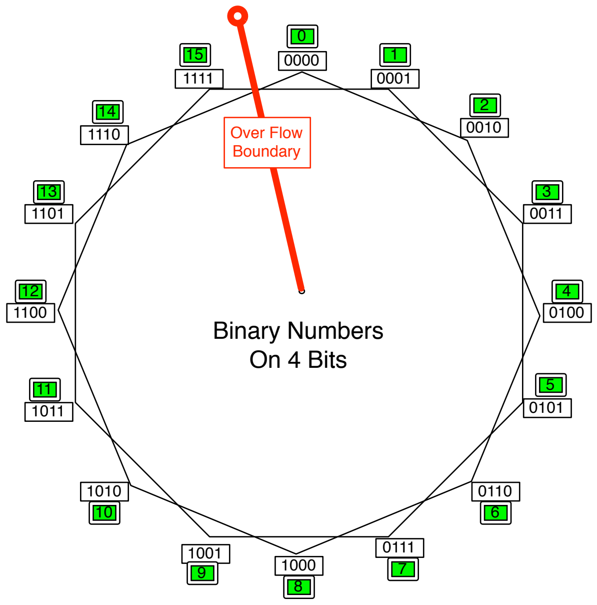


Figure 3.1: Binary Numbers on 4 Bits

to the binary representation of 15, namely 1111. The next number is 16, or in binary 10000. But we only have 4 bits, so the carry ”gets lost” and the result is 15+1 =0.

In fact, if we add any two binary numbers on 4 bits which produce a result greater than 15, the result will be incorrect , since the carry out of the 4th bit has no where to go. We say it ”overflows” the limits of our word size. This can be detected exactly when there is a carry out of the last digit of the word during an addition.

With a modern machine using 64 bit words we can represent much larger integers, since there are $2^{64} = 18,446,744,073,709,551,616$ different patterns. This range of values is usually sufficient for most everyday purposes. Nevertheless, if we wish to use very large numbers, we have to write software to use multiple words to represent very large values.

What if we want to use fractions? Well, we have several choices. One is to simply assumed that a certain number of bits represent the fractional part in all numbers. The other is to store the location of the radix point, perhaps as part of the word. One way of doing this is called floating point, but we will not discuss it in this part of the course.

3.1.3 Two's Complement

If we want to allow subtraction just as in BeeperBot we need to modify our representation to include negative numbers. In BeeperBot we placed a beeper to the left of a pile to indicate the pile represented a negative value. Similarly, we could for example say that if the left most bit is a 1 then the rest of the bits represent a negative number.

There is in fact a clever way of doing this that makes it easy to implement negative numbers and do addition and subtraction, as well as detect overflow conditions. In figure 3.2 we illustrate the *Two's Complement* representation on 4 bits.

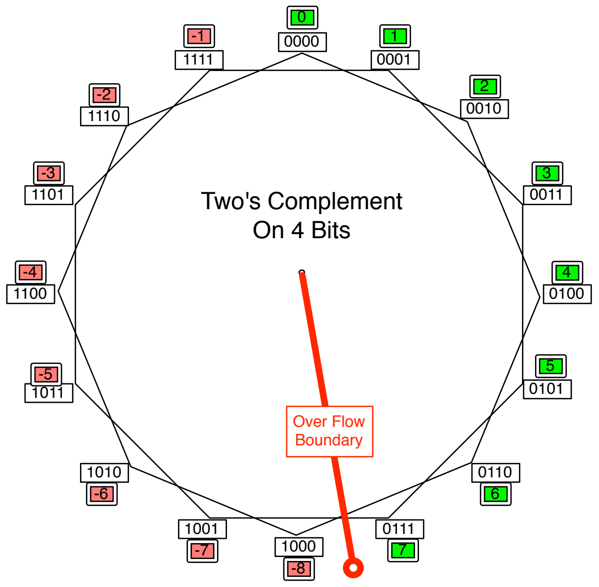


Figure 3.2: Two's Complement on 4 Bits

Notice that the bit patterns are arranged in exactly the same order as they are for the usual binary number representation. However, we change the meaning of the patterns when the leftmost digit is a 1. Starting with 0 at the top, and following a counter-clockwise order we get the numbers $-1, -2, \ldots, -8$. These negative numbers correspond to the bit patterns with a 1 in the leftmost bit. The other half of the bit patterns correspond to non-negative numbers. Since 0 is not considered a positive number, we have one less positive number than negative, for example here the largest positive number is 7, the smallest negative is -8.

So, everywhere except across the overflow boundary, moving clockwise increases the number by 1, and counterclockwise decreases the number by 1. The range of numbers is now $-8, -7, \ldots, 7$.

The non-negative numbers are represented by the same bit patterns as in standard binary. What about the negative numbers? Well, to find negative 5 for example, you could draw the wheel as in figure 3.2 and count down to -5. But for a 64 bit representation, and a number such as -1398745163 this might not be convenient.

Because we have the clockwise pattern set that corresponds to addition, we can find negative 5 by the following reasoning:

Two's Complement Negation Insight The bit pattern for $-X$ should be the set of bits which when added to the bit pattern for X gives us the bit pattern for 0.

Here X is any integer that can be represented in the two's complement form with the number of bits we are using.

Now comes the trick! Note that 1111 is -1, and in general the binary string of all 1's corresponds to -1 for any number of bits. If we add 1 to -1 we get 0. Suppose we have a number expressed in 8-bit binary as 01101010. *The one's complement* of a binary string is created by replacing each 0 by a 1, and each 1 by a zero. So the one's complement of that string is 10010101.

What happens if we add a binary number to its one's complement?

```
 01101010
 10010101
---------
 11111111
```

That's right, we get -1. To get zero we just need to add 1 more. (Remember, the carry on the last one just drops off the earth because we only have 8 bits.)

So how do we get the two's complement? Well, we just showed it is one more than the one's complement, so the rule is *find the one's complement, and then add 1.* So for the above string

```
01101010   original X
10010101   ones complement
       1   add one
--------------------------
10010110   two's complement
--------------------------
    Testing The Result
           01101010
       +   10010110
--------------------------
           00000000
```

This technique works for any two's complement number except $10\ldots00$. That is, to find the negative of a negative, you use the same rule, and get the representation of the positive. Why does it fail for the one case? Try it on the 4-bit representation of -8, 1000 and see! What is wrong with the answer you get?

To do subtraction when you have two's complement representation, you merely need to find the two's complement and do addition. This saves circuitry in the computer.

Finally, we need to detect when overflow occurs during addition. In ordinary binary number representation, we can check for a carry out of the last digit. But in two's complement this will not help, since this merely means we have switched from a negative to a non-negative value; for example try adding 3 to −2 in figure 3.2. Nor is it sufficient to check the carry out of the next to the last digit.

There is a simple way to check. First off, convince yourself that adding a negative and a non-negative will never cause overflow. You should be able to do this easily on the figure.

Next, notice that if I add any two non-negative numbers that cause overflow, then the result will be a negative number, and that if I add any two negatives that cause overflow then the result will be a non-negative.

Given these observations, the test is easy. If the two numbers to be added have the same sign, that is if the leftmost binary digit in both of them is either 0 or both 1, and the result is the opposite then overflow occurred. Otherwise, it is okay.

3.2 Binary Representation of Characters

Characters are represented in computers by using a particular bit pattern for each different character. ASCII is the most common, and is based on one byte per character, which means it can represent 256 different characters. Although this was sufficient for earlier times, with the inclusion of languages other than English, and the need for many control characters, special symbols etc. more flexibility is required. ASCII is now being supplanted by a new standard referred to as Unicode.

3.3 Boolean Logic

Boolean logic was developed by George Boole (1815-1864).

While working with BeeperBot we introduced *boolean statements*, statements which may take *true* or *false* values, depending on the state of BB's world. For example

- *next_to_a_beeper* has value *true* when BB is standing next to one or more beepers, and *false* otherwise.
- *has_beeper(2)* is *true* if there is one or more beepers in Bag 2, and *false* if there are none.

Boolean statements can also be defined in other worlds, for example "is wearing red" would be true for the person wearing red, false for someone who is not.

For generic references, we usually use a letter such as P,Q .. to represent a boolean statement.

For example, if n is a non-negative integer, then we could let P = "$(n > 0)$". Here is a table showing how P relates to n.

$n = 0$	$n = 1$ or $n = 2$, etc.
P is *false*	P is *true*

Note how this corresponds to *next_to_a_beeper* and *has_beeper(2)* .

The value *false* is often replaced by F, or 0 The value *true* is often replaced by T or 1. We will use 0 and 1 notation from now on.

Here is another example that you should think about carefully. Let n, m be arbitrary integers. We will call such letters *variables* when we start programming. Now consider the boolean statement "$(n < m)$', which we will designate by the letter P. Remember, 1 means *true*, 0 means *false*. We leave the last few entries as an exercise. If you do not see it, draw a number line.

n	m	P
1	2	1
2	7	1
6	2	0
-3	-5	0
-3	-2	?
7	-5	?

We can also have *boolean functions.* Given one or more boolean statements P, Q, a function f(P,Q,) assigns some value for each combination. The only *output* values we allow are again 0 and 1. We call P,Q etc. the *inputs.*

Our first example has f = **NOT**,and is a function of one input boolean value. We can write **NOT**(P), but usually write ˜P or $\overline{\text{P}}$.

We can express functions such as **NOT** using *truth tables.* Here the *input* is P, the output is ˜P.

Input P	Output ˜P
0	1
1	0

What are the other possible functions of one boolean input? Well, there are not that many, because we only allow 0 or 1 as output values. Here is a complete list or all possible boolean functions of 1 input.

Input	Output Functions			
P	F_0	F_1	F_2	F_3
0	0	1	0	1
1	0	0	1	1

Why is this a complete list? Well, there are only two inputs, so there are only two rows in the table. With only two rows, there can only be $2^2 = 4$ different patterns for the columns, each column representing one of the functions. Notice that the subscripts assigned to the function identifiers correspond to the integers represented by the functions values, e.g. $F_0 \leftrightarrow 00$, $F_1 \leftrightarrow 01$, $F_2 \leftrightarrow 10$ and $F_3 \leftrightarrow 11$.

We see that F_1 is the **NOT** function. The functions have the following naming conventions which should be apparent.

F_0 **FALSE**

F_1 **NOT**

F_2 **IDENTITY**

F_3 **TRUE**

Now let us consider functions of two inputs. The first is the **AND** function. Here is the definition of **AND** using a truth table.

Inputs		Output
P	Q	P **AND** Q
0	0	0
0	1	0
1	0	0
1	1	1

The **AND** function is true exactly when both of its inputs are true. We could write this as **AND**(P,Q), but usually prefer to write P **AND** Q, treating **AND** as an operator similar to multiplication in arithmetic, for example 5×6. In fact,you should learn the short hand that $\mathrm{P} \wedge \mathrm{Q}$ means P **AND** Q.

The second function of note is **OR**. It is defined by the following truth table.

Inputs		Output
P	Q	P **OR** Q
0	0	0
0	1	1
1	0	1
1	1	1

OR is true when at least one of its inputs is true. We could write **OR**(P,Q) or P **OR** Q, but mostly use the symbolic notation $\mathrm{P} \vee \mathrm{Q}$.

In table 3.1 we define all 16 boolean functions on two inputs. Because of the size we have had to rearrange this as a vertical table, with the inputs at the top, and a function on each row.

It turns out that any boolean function can be built from the functions **NOT**, **AND** and **OR**. We say that two boolean functions are *logically equivalent* if for any input set they agree on the output. To be more precise, for any boolean function, it is possible to build another boolean function composed of just **NOT**, **AND** and **OR** operators that is logically equivalent.

For example, the **NAND** function, F_{14} in table 3.1, is **NOT**(P **AND** Q) which using the standard notation can be written as $\overline{\mathrm{P} \wedge \mathrm{Q}}$. We can see this by creating a truth table. Here we present the result in stages. First we fill in

Inputs					
P	0	1	0	1	
Q	0	0	1	1	
Output Functions					Name
F_0	0	0	0	0	**FALSE**
F_1	0	0	0	1	**AND**
F_2	0	0	1	0	
F_3	0	0	1	1	Q
F_4	0	1	0	0	
F_5	0	1	0	1	P
F_6	0	1	1	0	**XOR**
F_7	0	1	1	1	**OR**
F_8	1	0	0	0	**NOR**
F_9	1	0	0	1	**EQUAL**
F_{10}	1	0	1	0	˜P
F_{11}	1	0	1	1	
F_{12}	1	1	0	0	˜Q
F_{13}	1	1	0	1	
F_{14}	1	1	1	0	**NAND**
F_{15}	1	1	1	1	**TRUE**

Table 3.1: All Two Input Boolean Functions

the result of P $\wedge$ Q from the truth table for **AND**. Then we find **NOT** of this result by applying the definition of **NOT** from its truth table to each of the results from the **AND**. Finally, we list the **NAND** definition from the table above so it is easy to compare and see the results are the same for each of the input value pairs.

P	Q	P $\wedge$ Q	˜(P $\wedge$ Q)	**NAND**
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

The **NOR** function, F_8in table 3.1, is **NOT**(P **OR** Q) which using the standard notation can be written as $\overline{\text{P} \vee \text{Q}}$.

For another example, consider the function **XOR**. We can implement this using the function composition $(\text{P} \vee \text{Q}) \wedge \overline{(\text{P} \wedge \text{Q})}$. Here is a truth table showing how the two compare. In this table we use the shorthand A = (P $\vee$ Q), and B = (P $\wedge$ Q). Thus, the entire function is $\text{A} \wedge \overline{\text{B}}$. We break the formula down into these parts, and compare to **XOR** as shown previously.

P	Q	A	B	$\text{A} \wedge \overline{\text{B}}$	**XOR**
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

To see how this works, consider the column headed A. Simply look up for each pair of input values the corresponding output in the definition of **OR**. Do the same using **AND** for the column headed B. For the column headed $\text{A} \wedge \overline{\text{B}}$ the inputs are A and B, but we have to apply **NOT** to B before using it. If this is unclear, you can insert a **NOT**(B) column and fill it in as an exercise. The last two columns are identical which completes our argument that the two functions are logically equivalent.

The **XOR** function is the same as **NOT**(P **EQUAL** Q). A good exercise in understanding is to reason through why $(\text{P} \vee \text{Q}) \wedge \overline{(\text{P} \wedge \text{Q})}$ is true exactly when P is not equal to Q. This may take you a while, but is well worth the effort.

Observe that if you consider the functions reading down from F_8 they are just the negations of the same functions reading up the table from F_7.

The exercises ask for further equivalences. In a subsequent section we show a generic construction technique for any boolean function on any number of inputs.

3.3.1 How Many Boolean Functions are There?

As we allow the number of inputs to grow, we get a very rapid growth in the number of possible boolean functions. Notice that for two inputs we got $2^4 = 16$ functions. Where did this come from? Well, the two inputs gave us $2^2 = 4$

different input pairs, namely $00, 01, 10$ and 11. Now for each of these pairs of inputs a function can output either 0 or 1. Since there are 4 pairs, that means the total choices for making a function is $2 \times 2 \times 2 \times 2 = 2^4 = 16$.

Consider for example the case where we have 3 inputs. With three inputs there are $2^3 = 8$ possible input patterns, corresponding to the binary representation of the binary numbers $0 \ldots 7$. Similar to the table for two inputs, we now have two choices, 0 or 1, for each of the 8 different input patterns, or $2^8 = 256$ different functions of three inputs.

In general, for n inputs we have 2^n different input combinations, and thus 2^{2^n} different functions. Table 3.2 has a short list illustrating how fast this grows as n increases.

Inputs	Input Combinations	Functions
n	2^n	2^{2^n}
1	2	4
2	4	16
3	8	256
4	16	65536
64	18,446,744,073,709,551,616	incomprehensible

Table 3.2: The Number of Boolean Functions on n Inputs

Although boolean logic is very simple at the outset, in some sense this enormous growth in possibilities with increasing number of bits is what allows it its computational power.

3.4 Boolean Circuits

The reason boolean logic is relevant to digital computers is that it is comparatively easy to build bi-state devices using electronic circuits. We will not develop the electronic level here. Instead we will look at logic gates that can be built from transistors and such.

Figure 3.3 shows six of the two input gates shown above. The first gate we

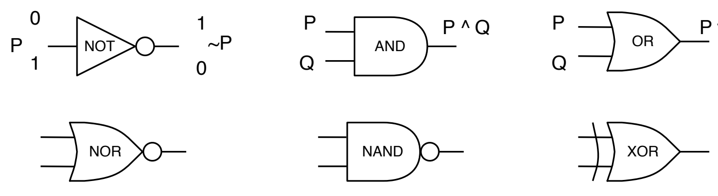


Figure 3.3: Boolean Gates

will consider is the **NOT** gate, shown in figure 3.3. As with all gates the inputs are shown on the left, and the result of the operation called the output is on the right. For the not gate, when we have a 0 as input, shown on the top left, we get a 1 as output shown on the top right.

In general to evaluate a circuit we start with certain inputs and follow them through the circuit to obtain the output. At each gate, we apply the logical definition of the gate to obtain the output. If we repeat this for every input combination, we can determine the complete function computed by the circuit.

A key observation is that for every boolean logic expression there is a corresponding circuit, and vice versa.

Let us consider the simple boolean function $\mathrm{P} \wedge \overline{\mathrm{P}}$. Note that in evaluating this, we first have to evaluate $\overline{\mathrm{P}}$ and then do the **AND**. To make this clear, let us state it in the equivalent form P **AND NOT**(P). We construct a circuit for this in figure 3.4. Notice how the input from P is split so that it can be input to

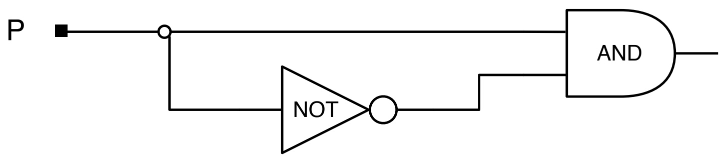


Figure 3.4: Gate Construction of P **AND NOT**(P).

two different gates. We can split any line as many times as required. However, we cannot merge lines, as the result would not be determinate. So, if we have two outputs we either treat them as two different functions, or we must combine them logically using a two input gate. In the example, the output of the **NOT** gate and the direct line from P are inputs to the **AND**.

Suppose input P=0. Then the output of the **NOT** gate can be determined from the truth table for **NOT** to be 1. Thus, the inputs to the **AND** gate are 0 and 1. We use the truth table of **AND** to find the output in this case should be 0.

Suppose input P =1. Then the output of the **NOT** gate can be determined from the truth table for **NOT** to be 0. Thus, the inputs to the **AND** gate are 1 and 0. We use the truth table of **AND** to find the output in this case should also be 0.

Thus, this circuit implements the one input function we called **FALSE**. If you want to think about it logically, it says that the statement "P is true and P is false" must always be false, whatever P is. Here is the corresponding truth table.

P	˜P	P **AND** ˜P
0	1	0
1	0	0

Figure 3.5 illustrates the circuit corresponding to the logic expression in the previous section for **XOR**, which is $(P \vee Q) \wedge \overline{(P \wedge Q)}$. Be sure you understand

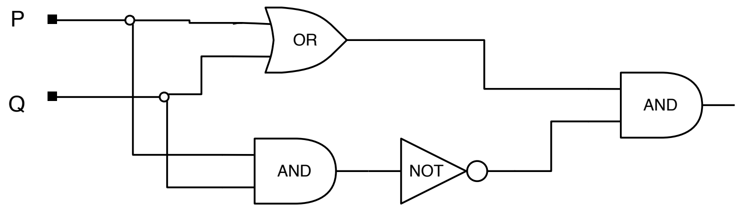


Figure 3.5: Gate Construction of **XOR**

how the circuit corresponds to the logical expression. For example, why is the **NOT** gate between the two **AND**'s?

Suppose P = 0 and Q=1. Then the inputs to the left most **AND** gate in figure 3.5 are 0 and 1, and the output of that gate is 0, according to the definition of **AND**. The inputs to the **OR** gate are also 0 and 1 and so the output of the **OR** gate is 1. The input to the **NOT** gate is 0, so its output 1. Now, since both inputs to the right **AND** are 1, its output is 1. Summarizing for inputs P=0, Q=1, the circuit outputs 1.

You should try the other three input pair possibilities to verify the output always matches the **XOR** as defined in table 3.1.

You should practice creating arbitrary boolean circuits, tracing them and converting them to logic formulae, and vice versa.

3.5 Multiple Inputs and Sum of Products Constructions

*** Stef's obs no motivation *** Suppose we have an expression like A **AND** B **AND** C **AND** D. Does it matter what order we evaluate this in? For example, consider (A **AND** B) **AND** (C **AND** D) versus A **AND** ((B **AND** C) **AND** D). Try it and see. What other orders can you find?

What you should learn is that for a sequence of **AND**'s such as this you get an output of 1 only if all inputs are 1's, and if any input is a 0 then the output is 0. Thus, we could write **AND**(A,B,C,D) as a multi-input **AND** function.

Figure 3.6 illustrates circuits for a three input **AND**. In the upper portion is shown how it could be built with two 2-input **AND**'s. However, it is more efficient to use a 3-input gate as shown in the bottom half.

Now consider the expression A **OR** B **OR** C **OR** D. Similar to the above you should find that the result is always 1 if any input is 1, and is 0 only when all inputs are 0. Again, it does not matter what order the inputs are evaluated in.

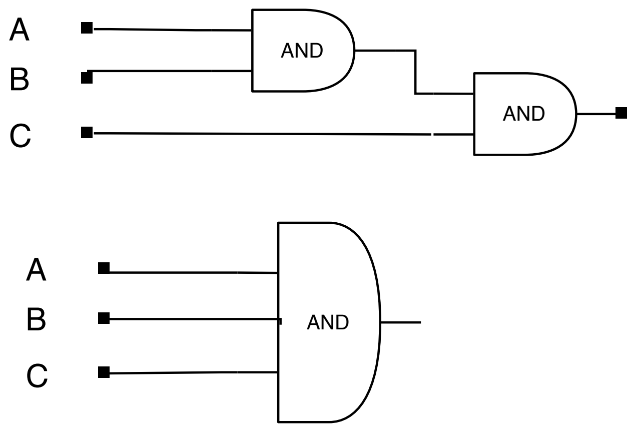


Figure 3.6: Three Input **AND**.

These observations extend in the obvious way to any number of inputs. The multiple input **OR** is defined to output 1 if at least one input is a 1, and outputs 0 only if all inputs are 0. The multiple input **AND** is defined to output 0 if at least one input is 0, and outputs 1 only if all inputs are 1.

Now consider the function F defined on three inputs by the following table. Note there is nothing special about F, it is just an arbitrary function chosen from the set of 256 possible ones on three inputs. We have added a label to the three lines where the function takes the value 1.

Inputs			Output
P	Q	R	F
0	0	0	0
0	0	1	0
0	1	0	1 A
0	1	1	0
1	0	0	1 B
1	0	1	0
1	1	0	0
1	1	1	1 C

Consider the labeled lines. We get F=1 if P=0, Q=1 and R=0 (A case), or if P=1, Q=0 and R=0 (B case), or if P=1, Q=1 and R=1.

We can express these cases logically as follows

A: $\overline{\mathrm{P}} \wedge \mathrm{Q} \wedge \overline{\mathrm{R}}$

B: $\mathrm{P} \wedge \overline{\mathrm{Q}} \wedge \overline{\mathrm{R}}$

C: $\mathrm{P} \wedge \mathrm{Q} \wedge \mathrm{R}$

F:

$$\mathrm{A} \vee \mathrm{B} \vee \mathrm{C} = (\overline{\mathrm{P}} \wedge \mathrm{Q} \wedge \overline{\mathrm{R}}) \vee (\mathrm{P} \wedge \overline{\mathrm{Q}} \wedge \overline{\mathrm{R}}) \vee (\mathrm{P} \wedge \mathrm{Q} \wedge \mathrm{R})$$

Now suppose we have a second function G defined by the following table.

Inputs			Output
P	Q	R	G
0	0	0	1 D
0	0	1	0
0	1	0	1 A
0	1	1	0
1	0	0	1 B
1	0	1	0
1	1	0	0
1	1	1	0

Note that this "shares" the A and B results used in F, and has one new line D. Thus, G = A **OR** B **OR** D.

We now illustrate the construction of a circuit to compute these functions, using a systematic layout method. First, each input is split into two lines, one negated with a not gate. Then, for each part A, B, C and D we simply need to

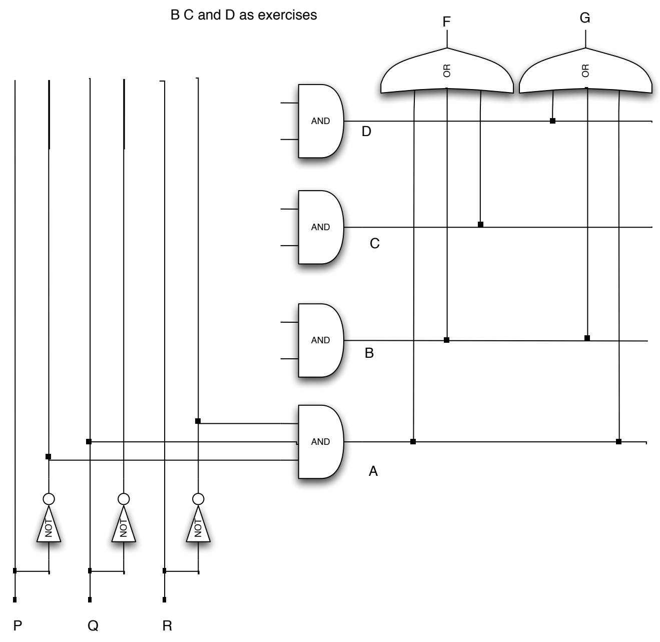


Figure 3.7: Circuit to compute F and G, constructed using the sum of products method. Connection of the inputs to B, C and D gates is left as an exercise.

connect the inputs that construct that sub-function. Then we **OR** the outputs of the **AND** gates together to get the functions F and G. This is illustrated in figure 3.7. This technique is called the *sum of products* method, and the regular circuit structure is called a gate array.

3.6 Binary Addition Logic

Review binary addition before reading this section.

We now look at the problem of binary addition as a boolean function. Consider adding one column of two binary numbers. There are the two digits, one from each number, lets call them A and B, and a carry in C_{in} which may be 0 or 1. For the first column, where we generally do not consider there is a carry, we can assume the carry in is 0.

We must compute two output functions, the bit sum S, and the carry out C_{out}. Table 3.3 gives the functions for adding two bits and a carry and producing the two outputs we desire.

Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.3: Addition Functions for Sum (S) and Carry

One of the exercises asks you to complete the sum of products circuit to compute the sum and carry. Note that this table is called a *full adder* definition because it includes the carry. Many texts and online sources refer to a *half adder*. This simply means the logic circuit is designed for adding two digits, but not adding the carry C_{in}.

Suppose we have a 3 digit number $A = 100$. Then we say the bits of A are $a_2a_1a_0$ which means $a_2 = 1$, $a_1 = 0$ and $a_0 = 0$. Similarly, we identify the bits b_2, b_1 and b_0 of the input B. Finally, we let the carries into each column be c_2, c_1 and c_0 where $c_0 = 0$ since the rightmost column has no carry in.

Let us now *abstract* the addition of one column by representing it as a box with inputs for each digit and carry. We illustrate this in figure 3.8. Each box is a copy of a completed circuit such as the one in figure 3.9. Notice that the carry out from one column is just the carry in to the next column. Thus the computation must proceed from right to left. If the last carry out is set to 1,

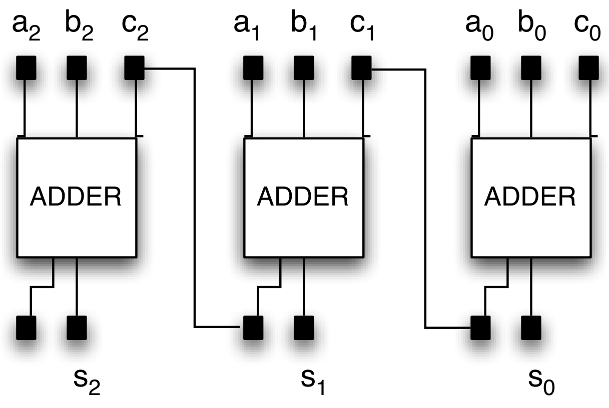


Figure 3.8: Ripple Adder for Two 3-bit Numbers.

then overflow has occurred on this 3-bit adder. The name "ripple adder" comes from the observation that the computation ripples from right to left.

Although this is sufficient to illustrate that logic is sufficient to do the computation of addition, in practice since each gate will have some delay in its operation, for computations on say 64 or 128 bit numbers the delay while all the carries propagate to the left might be unacceptable. Thus, in practice more complex but faster circuits will typically be designed.

3.7 Chapter 3 Questions

1. Use BeeperBot to simulate the **NOT** function. Let one cell represent the input, another the output. You let 0 be represented by 0 beepers and 1 be represented by 1 beeper. Now encode the following algorithm

```
If there is no beeper on the input cell, then
       Put a beeper on the output cell
Else
      Do nothing
```

2. Using a similar representation as above, write BeeperBot code to simulate the **AND** function.

3. Using a similar representation as above, write BeeperBot code to simulate the **OR** function.

4. Show the circuit for the expression $\mathrm{Q} \vee \overline{\mathrm{Q}}$. What function does it compute?

5. Show how each of the sixteen 2-input functions can be built using only **AND**, **OR** and **NOT** gates. HINT: to show this, you should show that for each of the 2-input functions there is a circuit consisting of only these gates that is logically equivalent. Note this shows that only these gates are required for any 2-input function.

6. Show that **NOT**(**NOT**(P)) = P.

7. **De Morgan's laws**
   Show the following two statements are correct, that is the left and right sides of each are logically equivalent.
   (a) $\overline{\mathrm{P} \vee \mathrm{Q}} = \overline{\mathrm{P}} \wedge \overline{\mathrm{Q}}$
   (b) $\overline{\mathrm{P} \wedge \mathrm{Q}} = \overline{\mathrm{P}} \vee \overline{\mathrm{Q}}$

8. Show how the **AND**, **OR** and **NOT** gates can be built using only the **NAND** gate. HINT: Use the previous two exercises.

   Note combined with the previous exercise, we see that **NAND** gates are sufficient for computing all two input functions. However, such a construction might not be the most efficient.

9. Consider the diagram in figure 3.7. Now suppose we had more functions defined on these three inputs that we wanted to compute simultaneously with F and G. What is the maximum number of **AND** gates we would ever need? Explain.

10. Table 3.3 defines the two functions required for binary addition. Complete the sum of products circuit in figure 3.9 for addition, where $c = C_{in}$ and $c' = C_{in}$ in the figure.

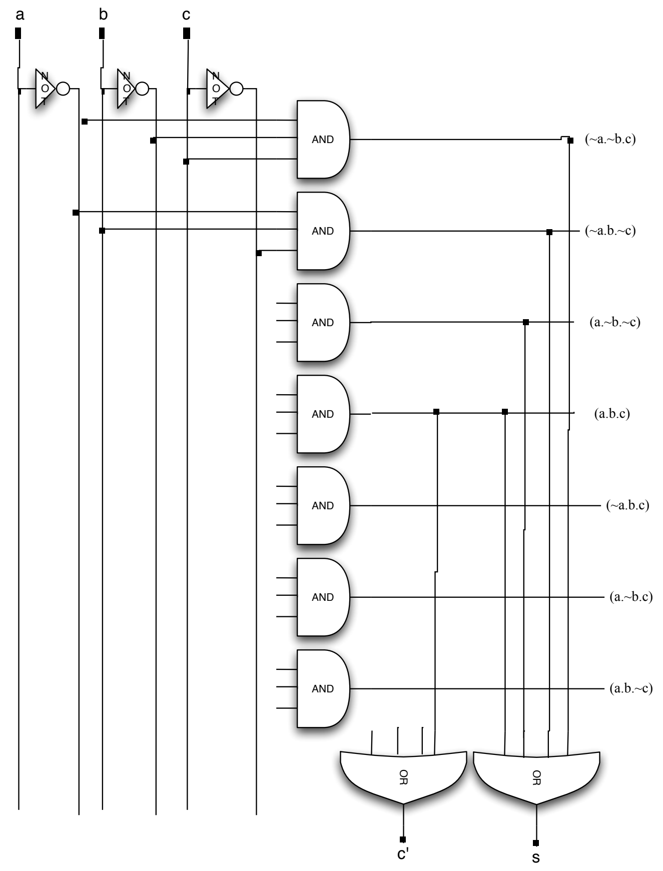


Figure 3.9: The Full Binary Adder: Complete as Exercise.

Chapter 4

Program Exploration Tool (PET)

In this chapter we discuss the language used in the Program Exploration Tool (PET), and briefly the model underlying it and how it relates to typical computer architecture. PET uses a language similar to many modern languages, but has the advantage of a simpler syntax, and it displays the computation in a step by step manner that should help in understanding what is going on.

In chapter 2 we presented the following summary of learning to program, in particular as applied to BeeperBot

1. Learning the language: For BeeperBot this should only take a short time because the language is very elementary.

2. Learning how to use the language to do elementary "mechanical" tasks: In Chapter 1 we show how move piles of beepers and the exercises and labs lead you to develop algorithms to follow paths of beepers, or sweep all beepers from an enclosed space.

3. Learning how to represent and model real world problems: In chapter 2 we discussed representing and displaying text, and the more difficult task of representing numbers and doing elementary arithmetic.

One of the struggles of modern computing science is the development of better and more powerful languages. In general, languages have been developed that try to minimize the effort of steps 2 and even 3, by building more into the language itself. This has the side effect of making the languages themselves ever more difficult to learn and use. Achieving the right balance is difficult, and the appropriate choice of language for any problem depends on the problem. For many specialized tasks, such as statistical analysis, or scripting in game programs, or mathematical programming specialized languages have been developed. But many general purpose languages such as JAVA and C++ have also been developed, and the result is there are many programming languages.

The Program Exploration Tool uses essentially a simplified subset of the language C. But before we look at the language, we need to add a 0th step to learning to program, and that is we need to understand the computational model over which the language operates. And before we do that, we will look very superficially at the general architecture that is used in todays digital computers. This will serve two purposes, one to acquaint you with how modern computers work, and two to provide an intuition for the slightly more abstract RAM model that PET can be seen to work on.

4.1 Architecture

Computer architecture, a term coined by Fred Brooks back in 1962, is a study of how to design computer systems. There are several levels of design. At the highest level is the system level. It involves how the different components are connected. These components include: memory, CPU, which stands of Central Processing Unit, and buses, which are for connecting different components together. The CPU is the brain of the whole system, without which there is no computer.

Each type of CPU speaks its own language. For example, the CPU used in a PlayStation is a cell processor from IBM and it speaks a different language than the CPU that you have on your PC or laptop. The language used by a CPU is commonly called the Instruction Set Architecture and forms an interface between the internal hardware of the CPU and the external world. This is crudely similar to languages we use as humans in communication, which form an interface between our own internal world and the external environment.

The design of such a language or interface forms the middle level of the architecture. The last level is commonly called the computer organization, which describes how the hardware can support the implementation of the instruction set of the CPU.

4.1.1 The von Neumann architecture

All contemporary computer systems adopt the following configuration.

Figure 4.1 illustrates this basic design. There are variations of this configuration but the general principle is the same.

When you look at your laptop, you probably think "computer". What you are actually looking at is a computing system. You will see a screen, where results of computations are **output** for you to view. You may attach a printer, which also takes output and presents it in a human viewable form.

You will see a keyboard which takes what you type and **inputs** it to the computer. You may also plug in a camera to input photos you have taken, or a microphone to input sounds.

You may input data, such as music or games, from a DVD or CD, and of course you may also be connected to the internet from which you can download (i.e. input) more data. You may attach various other media, such as external

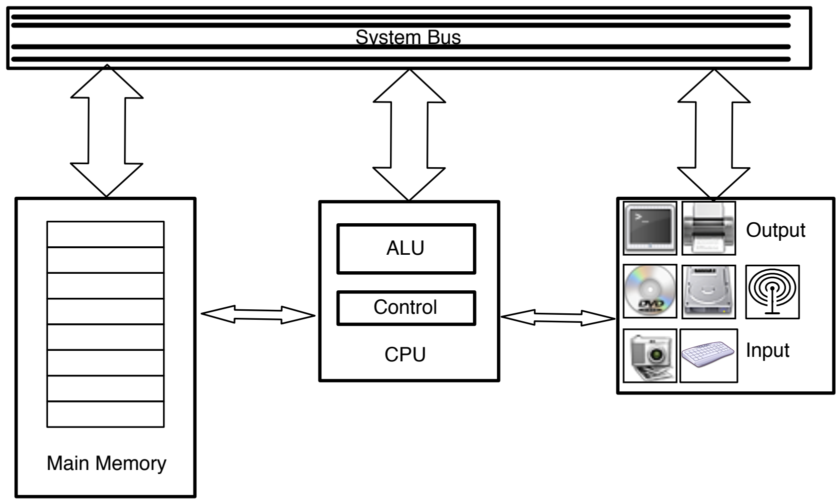


Figure 4.1: von Neumann Architecture

hard drives, and flash memories etc. Of course, you can also output data to these devices, so they are legitimately seen as both input and output devices. To be fair, some screens are touch screens and so also act as input devices and you sometimes can move data onto the memory of a camera from your computer, so they may occasionally also be considered as output devices.

But where is all this I/O (input/output) going? Well, it is going to the main memory, or to the internal hard drive, which for our purposes is just a slower but bigger and more persistent extension to the main memory. In our simplistic model, we suppose that it all is processed through the CPU (central processing unit), although some transfers may be handled by special device handlers.

The system bus pictured in figure 4.1 is the set of connections that acts as the principle connection between the devices, memory and the CPU. You can see vestiges of this in the various holes in the sides of your laptop which allow you to plug in cables, thereby connecting the bus to other devices.

Memory is best thought of as a very long sequence of words, as illustrated in figure 4.2. As we mentioned previously each word has some fixed number of bits, 4 in the illustration, but typically 32 or 64. Each memory cell has a particular address. In the illustration addresses are given in base ten for illustration purposes to separate them from the data (in reality they are also specified as binary in the machine).

We will discuss somewhat more the internals of the CPU and how programs are processed in section 4.2.

4.2 Simplified Model Underlying the Program Exploration Tool

For BeeperBot, the computational model is fairly simple and transparent. BB operates in a two dimensional world established on a rectangular grid. Grid cells may contain any non-negative number of beepers. BB can detect whether or not the cell it is in contains beepers, and has some local memory capacity in terms of 10 bags that may hold any number of beepers. The programming language has two classes of statements; those that interact with the world or the bags in some manner, and those that control the execution of the program such as *if* and *while*.

PET operates on a computational model that is more similar to a modern computer. To understand it, we will extract the CPU and memory units from figure 4.1 and expand them somewhat in figure 4.2. But be aware, that PET like any real programming language still allows keyboard input and produces output, so the I/O units are implied even if not shown in figure 4.2.

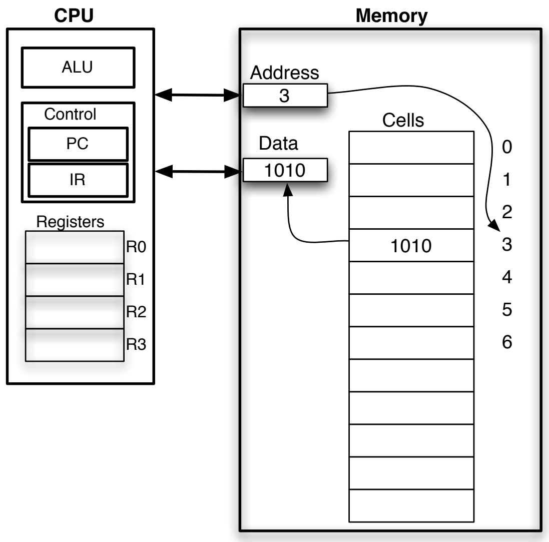


Figure 4.2: Random Access Memory (RAM) model

First we will give a brief explanation of each of the labels in the figure. To find out more, look in the “links.html” file associated with these notes.

As already mentioned CPU means the central processing unit. It is the heart of the computer, and the point at which computation occurs. Within the the CPU, the ALU, or Arithmetic Logic Unit, contains the circuits that implement operations such as addition, subtraction and multiplication, as well as logical operations. This then is where the circuits from the previous chapter fit in.

The control unit is where the actual machine code is taken from memory and set up for execution. The program counter (PC) indicates the address in memory of the next instruction. The actual instruction is copied from memory to the instruction register (IR).

A CPU may contain an additional set of registers, which are on-chip very fast memory cells. Because they are part of the CPU, they can be accessed directly by the ALU without going through the bus to the memory interface, and thus operations on them are very fast. Our fantasy processor has 4, labeled R0 through R3.

The memory unit contains cells, each of which can be individually addressed. Here we just assume a cell is one word. In addition, our simplified model has two special registers. One is the memory address register and the other is the data register.

So how does a machine language program work? First the language itself consists of instructions encoded in binary that can be directly executed by the CPU. A program will be located starting at some address in memory. This address will be loaded into the PC. Then the control unit begins to repeat the following sequence of operations, known as the *fetch, decode, execute* cycle.

1. Fetch an instruction. The address contained in the PC is sent to the address register in memory along with a signal that the current data is wanted. The memory module loads the data from the addressed cell into the data register. The control then puts this into the instruction register. The PC is incremented to indicate where the next instruction is.

2. Decode the instruction. Since an instruction consists of different parts, called fields, each field carries a particular instruction for the CPU to perform. To obtain each field is to decode the instruction.

3. Execute the instruction. If the instruction was arithmetic or logical, then the ALU performs the instruction. If the instruction was a programming branch instruction, then the PC may be changed. If the instruction was a load instruction, then data will be copied from some memory location to a register. If the instruction was a store instruction, then data will be copied from a register to memory.

Different CPUs will implement different instruction sets, and so there is a significant variety in machine level languages. But in general there will be something akin to the following.

Suppose we want to add two numbers in memory and store the result in a third cell. A typical chip design might have instructions such as

- LOAD R0, Addr
- STORE R2, Addr
- ADD R0,R1,R2

The first of these copies a number from the memory cell at address Addr into the register R0. The second copies the value in register R2 to the memory cell at address Addr and the third adds the numbers in the first two registers and puts the result into the third. Of course these instructions will also work on different registers. In our example we have R0, R1, R2 and R3.

The machine code for LOAD might be 0101. Since there are 4 registers, the next 2 bits could be the register address, and the remaining bits would be the memory address. Say the first memory is 1001, and for sake of illustration we have 10 bit words. Then the instruction LOAD R0, 1001 in binary would be 0101001001. There would be similar patterns for other instructions.

The complete program to do the addition might be

```
LOAD R0, 1001
LOAD R1, 1010
ADD R0, R1, R3
STORE R3, 1011
```

Note that each line occupies a word of memory. This program might be stored at memory locations 0011 through 0110. To execute it, the PC would be initialized to 0011.

So, how does this relate to a PET program? Well, each instruction in a PET program is translated to a set of machine instructions. Suppose we have the following PET program, where the dots indicate other code (that assigns values to the variables B and C e.g.)

```
int A, B, C
define main {
      ....
   A = B + C
}
```

First, note that A, B and C are *variables* and this means that they are merely labels that will be assigned addresses by the PET program when it runs or compiles (addresses can be assigned at compile time or run time depending on the system). Suppose B is assigned the address 1001, C is assigned the address 1010 and C is assigned the address 1011. Then the line `A = B + C` will result in a machine level code sequence just like the one we had above.

The key lessons with respect to PET

- Every PET line results in a few lines of machine level code
- Variables in PET correspond to memory cells; variable names are simply labels for memory addresses
- All of the machine code is carried out step by step in the CPU

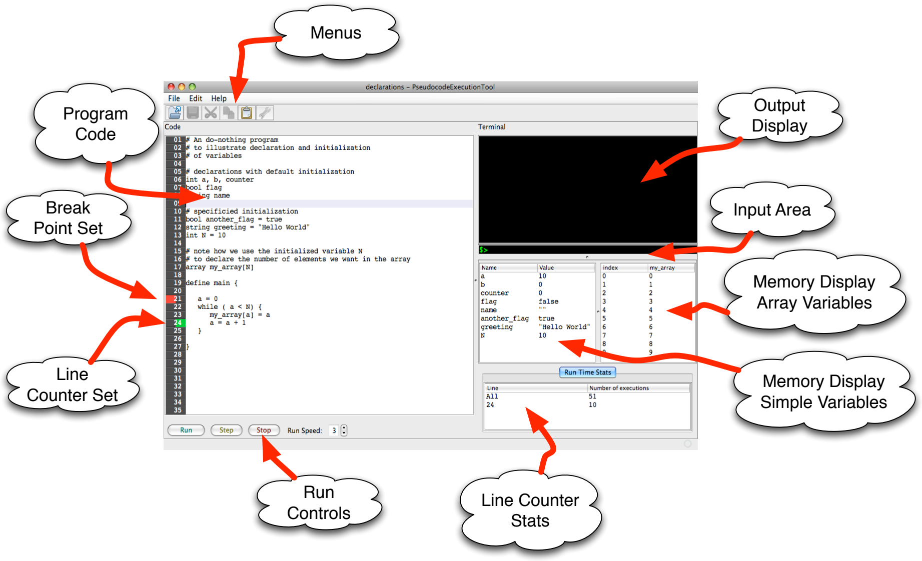
And finally, every logical and arithmetic function, all of memory, all of the control counters and all the registers are built using Boolean gates and circuits.

4.3 PET Language

It is time to begin the first step of learning a new programming language as listed in the introduction, namely to consider the features the language or tool provides. In this case you are lucky, because the instructors have chosen to provide you with a fairly simple to learn programming language which is also executed inside a separate user interface panel.

4.3.1 Interface

Figure 4.3 identifies some of the main features of the PET interface. These features are self-explanatory or will be referenced in subsequent sections, so we provide no overview here.



```
# An do-nothing program
# to illustrate declaration and initialization
# of variables

# declarations with default initialization
int a, b, counter
bool flag
string name

# specificied initialization
bool another_flag = true
string greeting = "Hello World"
int N = 10

# note how we use the initialized variable N
# to declare the number of elements we want in the array
array my_array[N]

define main {

   a = 0
   while ( a < N) {
      my_array[a] = a
      a = a + 1
   }

}
```

```
$>
```

Name	Value
a	10
b	0
counter	0
flag	false
name	""
another_flag	true
greeting	"Hello World"
N	10

index	my_array
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Line	Number of executions
All	51
24	10

Figure 4.3: PET Interface

4.3.2 Comments

Just as in BeeperBot, a comment starts with the symbol # and continues to the end of the line. All text in a comment is ignored when the program is run. A side effect of the simplicity of PET is that you cannot get the program to print a # symbol.

It is a good idea to get in the habit of commenting your code. Remember, a TA has to understand and mark your code. Anything you can do to help the TA understand what you are trying to do is likely to increase your mark.

4.3.3 Variables

The concept of a variable was touched on in section 4.2. A *variable is a named collection of memory cells.* In PET, there are two classes of variables, simple variables and compound variables.

Simple Variables

Simple variables hold only one value at any time. It is easiest to think of this value as occupying one memory cell, although this may not be strictly true with respect to the underlying machine.

From the course so far, you should realize that the values that any variable holds are strings of binary digits. But we do not usually want to work with values at this level. So PET provides built-in *data types.* For our purposes, a data type expresses what kind of information the bits are supposed to represent.

The three types of simple variables are

int The variable stores an integer, which may be positive or negative. The value will be displayed in base ten format, both in the memory viewing pane, and in the input and output panes.

bool The variable has only two values, representing **false** or **true**. These are the values that are displayed.

string The variable can only be assigned a string of characters enclosed in double quotes.

Compound Variables: Arrays

Compound variables are, as the name suggests, collections of simple variables under one name. Each part of the variable may be assigned or copied individually. In PET, the only such variable is an array of integers.

array The variable will be an array of integers, with the number of integers in the array specified by $[n]$, where n is an integer.

Declaration and Initialization

Here are the rules for declaring variables in PET

- All variable declarations must precede the first **define** statement.
- Variable names may only contain upper or lower case letters and the under bar character "_".
- Declarations start with one of the key words **int, bool, string, array** and are followed by one or more variable names separated by commas. Note: a bug allows only one string declaration per line.

- **int, bool** and **string** variables may be given initial values using the symbol "=". Array variables can only be initialized by default (to 0).

By default, all **int** variables, including arrays, are initialized to the value zero, all **bool** variables are initialized to **false** and all string variables are initialized to the null string, designated by " ". Please note: assigning default values is a feature of PET which is not found in most languages. It's good practice to get into the habit of initializing your variables and not depend on a default setting.

You can override the default initializations for **int, bool** and **string** variables by using the assignment operator "=" followed by a value of the appropriate type.

The following code illustrates precisely all the modes declaring variables, and initializing them to either default or preferred values. If you run this code in PET, you can examine the effects of initialization in the memory display panes of the interface. The code is in the file called *declarations.txt* in the sample programs.

```
# declarations with default initialization
int a, b, counter
bool flag
string name

# declarations with user initialization
int N = 10, M = N + 5
bool another_flag = true
string greeting = "Hello World"


# declaration of array using initialized N
array my_array[N]
array array_b[M]

define main {

}
```

There is a special command for filling an array with random numbers called **fillarray**. It takes the name of the array as an argument. For example, we could fill the arrays in the previous code by changing the **define main** routine as follows

```
define main {
   fillarray my_array
   fillarray array_b
}
```

The numbers inserted are in the range $[1 \ldots 10 \times size]$ where $size$ is the size of the array. Thus, *my_array* would have numbers in the range $[1 \ldots 100]$ and *array_b* would have numbers in the range $[1 \ldots 150]$.

This command is used to create random arrays for sorting or other purposes, since input of a larger array is very time consuming in PET.

We will find that **filllarray** is also very handy for generating random numbers for certain games we will be programming.

4.3.4 Expressions and Assignment

First let us consider simple variables. In code a statement such as

```
a = b
```

is called an *assignment* statement, where "=" is the assignment operator. The effect of this statement is to copy the value of the variable `b` into variable `a`. The value of `b` is not altered.

To work correctly, `a` and `b` must be of the same type, that is either both **int** or both **bool** or both **string**.

For array variables, you must assign each element of the array independently. To select an element you use an integer called the **index**. If the array was declared with `N` elements, then the set of indices are $0, \ldots \mathtt{N} - 1$. Since the elements are integers, you can assign only values of type **int** to the elements of the array. Here are some examples, where you can assume `a, b, c` are integers and X is an array with 10 elements (indexed $0 \ldots 9$).

```
X[1] = 5
X[5] = a
c = X[b]
X[c] = X[b]
```

Note that when a variable of type **int** is used to index an array as in some of these examples, an error will occur if the index is negative or greater than or equal to the array size `N`.

Variables of type **bool** can only have the values **true** or **false**. We can also copy Boolean values using assignment.

String variables can be assigned any sequence of printable characters, except # and some control characters, using an assignment statement with the string on the right hand side contained in double quotes. String variables can also be copied using assignment.

If all we could do is assign or copy variables, our language would be of limited use. However, we can also evaluate expressions.

There are two types of expressions available in PET, arithmetic expressions and Boolean expressions.

Arithmetic Expressions

Arithmetic expressions in PET can be created with the *arithmetic operators* `+, *, -, /` and *operands* that are integers, integer variables or integer array elements. Also, you can use `( )` to indicate how the expression is to be evaluated. Here are some examples, where all non-array variables are assumed to be of type **int**.

```
a = 3*5
b = (a + 5)*7
Y[3] = (X[5] + 9) - Y[2]
X[a] = X[a] + X[a]*2 + 1
c = 7/3
```

Consider the second last of these expressions. Suppose the `a`'th element of `X`, `X[a]` has value 5 before this statement is executed. Then after it executes the value of `X[a]` will be 16. Note that the right hand side is completely evaluated before the value of `X[a]` is changed.

The other example of note is the last line. We have only integer arithmetic in PET, and so in this division the fraction is lost. After executing this line, the variable `c` would have the value 2.

Boolean Expressions

So far we have been using the terms Boolean statement to mean a statement that has one of two values, **false** or **true**, (or 0 or 1). We have so far been rather loose in our terminology, but now must be a bit more specific.

A *Boolean expression in PET* is an expression that evaluates to either **false** or **true**. The *Boolean operators* used in PET come in two flavors, *logical operators* and *comparison operators.*

Logical operators These are **and, or** and **not**. The operands that these operate on must evaluate to one of the Boolean values **false** or **true**.

Comparison Operators These are

- `==` meaning "is equal to". The operands of this can be of type **int, bool** or **string**, but both operands must be of the same type.
- `>, >=, <, <=` meaning respectively "greater than", "greater or equal", "less than" and "less or equal". These operators can only be applied to operands of type **int**.

Here is a sample program called *expression.txt* that is available in the sample programs illustrating construction of various expressions. You should run this program step by step in PET and observe what the result of each assignment is.

```
# ILLUSTRATION OF EXPRESSIONS, ASSIGNMENT
# STEP THROUGH THIS PROGRAM IN STEP MODE
# AND AFTER EACH STEP OBSERVE THE VALUES THAT CHANGED
# MAKE SURE YOU UNDERSTAND EACH RESULT

# some ints
int a, b, c

# Boolean values
```

```
bool bx, by, bz, bw, bv, bu

# strings
string sn
string sm
string sp

# declaration of array using initialized N
int N = 10
array X[N]

define main {

# assign some values to ints
  a = 5
  b = 6

# assign some values to bools
  bx = true
  by = false

# assign strings
  sn = "Yes"
  sm = "yes"
  sp = "yes"

# arithmetic expressions
  c = (a + b) / 3

  X[a] = ( c * (b+1)) / ( 2+5)

# Boolean expressions
# using ony Boolean variables
  bz = bx and not by

# asking Boolean questions about integers
  bw = (c+2) >= a

# asking are two strings equal
  bv = sn == sm

# again -- why is bu true and bv false?
# hint: capital Y
  bu = sm == sp

# compund question about arithmetic values
```

```
  bx = (a < 5) and (3 == X[5])

# maybe you want to ask is a == 5 or 3?
# wrong way
# by = a == 5 or 3
# right way
  by = (a == 5) or ( a == 3)

# the following will cause an error
# if it is uncommented - why?
#  bx = bx and not a < 5

# the following is okay - why?
  bx = bx and not (a < 5 )

}
```

4.3.5 Control Statements and Subroutines

Control statements are PET commands that cause the execution of the code to change from the normal "execute the next line" mode. In PET there are three control statements

if . . . else This looks very much like the **if . . . else** in BeeperBot, including that the **else** part is optional.

```
if (conditional) {
      . . .
} else {
      . . .
}
```

The *conditional* in PET differs from BeeperBot in that it can be any Boolean expression or a variable of type **bool**. For example, the Boolean expressions in the preceding section can be used as the conditional in any **if** statement.

while This looks very much like the **while** in BeeperBot.

```
while (conditional) {
      . . .
}
```

The *conditional* in PET differs from BeeperBot in that it can be any Boolean expression or a variable of type **bool**. For example, the Boolean expressions in the preceding section can be used as the conditional in any **while** statement.

call In BeeperBot a subroutine is called by simply putting its name as a program line. In PET, you must use the keyword **call**. As in BeeperBot, subroutines are defined using the keyword **define**. Here is an example

```
define my_sub {
        writeline "Hello World"
}

define main {
        call my_sub
}
```

Compared to languages such as C or Java, PET subroutines are very primitive. They do not provide parameters, or local declaration of variables etc. This gives us the opportunity to discuss these ideas from the "wouldn't it be nice..." perspective.

4.3.6 Input and Output

There are two output statements, **write** and **writeline**. Either takes a sequence of variables or strings in quotes separated by spaces and prints these arguments on the same line usually separated by a space. (A bug? Try the program below to see space not always appended.) A series of writes will continue adding to the same line of output, while **writeline** starts a new line after printing its arguments. You can only output arrays element by element using indices. All output goes to the pane named "Terminal" and labeled "Output Display" in figure 4.3.

There is one input statement, **read**, which can only be used with the keyboard. It takes exactly one argument of type **int, bool, string** or one indexed element of an array. Keyboard input is entered in the pane labeled "Input area" in figure 4.3. Note: you must press the "enter" or "return" key after entering your value to complete a **read** command.

The following program is called *readwrite.txt* and can be found in the sample programs.

```
# simple input and output of bool, string
# and a small array

bool x
string name
int i

int N = 10
array A[N]

define main {
   writeline "Enter true or false"
   read x
   writeline "you entered" x
   writeline "Enter some text"
   read name
   writeline "You entered" name
   i = 0
   while ( i < N ) {
        read A[i]
        i = i+1
   }

   # for variety, we print the
   # array in reverse order
   i = N-1
   while (i >= 0) {
     writeline "Element" i "is" A[i]
     i = i-1
   }
}
```

4.3.7 Controlling and Monitoring Code Execution

In the area labeled “Run Controls” in figure 4.3 there are three buttons and a speed control. *Run* starts the process executing at the indicated speed, while *Step* advances the process one program step on each click. Speeds can be set from 0 to 5, where at speed 5 the tracing is turned off, and, for most programs you will write, the execution will appear to be instantaneous. *Stop* terminates the process. Note that the process must be terminated before you can edit your code.

Two additional features of the PET interface are very useful in debugging and analyzing your programs. These are indicated in figure 4.3 by the labels “Break Point Set” and “Line Counter Set”. To activate these, select the line number of the code line you want the effect to take place on, and right click (control-click if you have a one button mouse). You can then toggle on or off the intended action.

If you select and click on “Toggle Statistics Line” the line will have a green highlight. When your program runs it will count how many times that particular line executes. This number is reported in the pane titled “Run Time Stats” and labeled as “Line Counter Stats” in figure 4.3. Selecting the same toggle a second time will turn this statistic off. You may turn the stats on for any set of lines, and they will all be reported. By default, the total count of all program steps is always displayed in this window.

If you select and click on the “Toggle Breakpoint” then the line number will be flagged with a red highlight. When you run the program it will halt whenever it reaches this line. You can change the speed if you wish. You can continue the run by hitting *Run*, or you can continue the process one step at a time by using the *Step* button. This feature is very useful for debugging. You may have a program that makes many steps before hitting a runtime error. If you set a breakpoint appropriately, you can let the program run at full speed until it hits the likely problem spot, and then step it one step at a time to see exactly what goes wrong.

4.3.8 Saving and Loading Source Code

PET provides for saving and loading programs using the *File* menu or the buttons located just under it. The interface is fairly standard, and documented in the built in help file.

4.4 PET Questions

These questions are designed to acquaint you with the PET language, and test your understanding.

Many of the questions are trace questions. Resist the urge to simply find the programs in the sample program sets and run them to find the answers. Instead you should run the programs "on paper" by tracing the variables and how they change. Only after you have obtained what you think is the correct answer should you verify it by running PET.

Remember, on the exam you will not have PET, and if you rob yourself of the chance to learn by taking shortcuts to get the answer, then you are the loser.

Most people will draw some sort of open ended table, with a column or row for each variable, then trace the effect on the variables as each statement is executed. The exact format of your trace is not important, as long as it makes it clear to you what is happening. And when you get the wrong answer, it is important that you have a trace clear enough that you can determine where you went wrong when you compare your results to those obtained from stepping through the program using PET.

1. Suppose a and b are declared as **int** and $flag$ is declared as **bool**. Which of the following PET statements will assign **true** to $flag$ regardless of the integer values assigned to a and b? Which will always assign **false**? Which if any are sometimes true and sometimes false? Note for integers a and b there only three cases; either `a == b` or `a < b` or `a > b`. Use this to establish truth tables to answer the following.

   (a) `flag = (a == b) == ( ( a < b) or (a > b) )`

   (b) `flag = ( not  (a == b)) == ( ( a <  b) or (a > b) )`

   (c) `flag = (a == b) == ( ( a <= b) and (a >= b) )`

   (d) `flag = (a <= b) == not ( b < a )`

   Note: in PET "`not (a == b)`" can also be written as "`a not = b`". Note that the space between the "not" and the "=" is required.

2. Will the following program halt? Explain its behavior in terms of how integers are represented. How many bits are used?

```
int a, b
define main {
    a = 1
    b = 0
    while ( not (a == 0) ) {
        b = b + 1
        a = a * 2
        writeline  a
   }
}
```

3. Trace the following program and indicate what the contents of the array A are. Check using PET.

```
int a

define second {
write "Hello "
 a = a + 2
}

define first {
  a = a * 3
  call second
  writeline "There!"
}

define main {
   a = 1
   call first
   writeline "a = " a
}
```

4. Trace the following program and indicate what the contents of the array A are. Check using PET.

```
int i
int N = 10
array A[N]

define main {
   i = 0
   while ( i < N ) {
     A[i] = N-1-i
     i = i+1
   }
   i = 0
   while ( i < N ) {
        A[i] = A[A[i]]
        i = i + 1
   }
}
```

5. In the following there are 4 different target statements. You need to find for each target a pair of integer values for a and b, that will result in the output of the target. Do not forget that integers can be negative or 0. If some targets are impossible, explain why.

```
int a, b, c
int i
bool flag

define main {
   writeline "Enter integer value for a "
   read a
   writeline "Enter integer value for b "
   read b

   flag = a < b

   c = a * b
   writeline c

   if ( flag and (c < a) ) {
       writeline "TARGET One"
   } else {
       if (flag) {
         writeline "TARGET Two"
        } else {
             if ( c < a ) {
               writeline "TARGET Three"
             } else {
                writeline "TARGET Four"
             }
        }
   }
}
```

6. Consider the following code and answer the three questions below.

```
int A, B

define main {
 read A
 B = 5

 while (A < B) {

  while (A < B) {

    A = A * 2
    B = B + 1
  }

  while (B < A) {

    B = B * 2
    A = A - 1

    if (B > 10) {
      B = 0
    }
  }
 }

 write A B
}
```

(a) Assume the user enters 2. Trace the contents of A and B for every change that is made and state what the output is when the write statement is executed.

(b) Repeat the trace, but this time assume the user enters 3.

(c) What happens if the user enters 0?

7. Trace the following algorithm.

```
int N = 21, i, k
array tag[N]

define main {
  i = 0

  while (i < N) {
    tag[i] = 0
    i = i + 1
  }
  i = 2

  while (i < N) {

    if (tag[i] == 0) {

      write i
      k = i*2

      while (k < N) {
        tag[k] = 1
        k = k + i
      }
    }

    i = i + 1
  }
}
```

(a) What is the output of this program?

(b) In general, what numbers will be printed by this program for any given N?

This method is known as the Sieve of Eratosthenes

Chapter 5

Algorithms and PET

In this chapter we will use the Program Exploration Tool (PET) to develop and explain some common algorithms. Typically we will start with some problem, write some high level pseudo code, then translate that into PET and debug and run it. For some of the cases, we will leave the programming and testing as exercises for the student.

By the end of this chapter you should have a grasp of some of the elementary algorithm design and analysis issues that a typical computing scientist would face.

5.1 What is an algorithm?

What exactly do we mean by the word *algorithm*? Here are some definitions

Encarta: A logical step-by-step procedure for solving a mathematical problem in a finite number of steps, often involving repetition of the same basic operation.

Wikipedia: An algorithm is an effective method for solving a problem using a finite sequence of instructions.

The Free Dictionary: A logical arithmetical or computational procedure that if correctly applied ensures the solution of a problem.

Merriam-Webster: A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly : a step-by-step procedure for solving a problem or accomplishing some end especially by a computer.

Church-Turing Thesis: Intuitive notion of algorithms equals Turing machine algorithms. (from M. Sipser, *Introduction to the theory of computation*, Chapter 3.3).

One we sort of like: A finite, ordered set of executable statements that defines a terminating process that solves some problem.

The dictionary definitions are too vague for our purposes.

The Church-Turing thesis provides a precise definition in terms of a Turing Machine. A *Turing machine* is formal mathematical model of computation which you can learn about from many sources, but is beyond the scope of this course. The precise definition of algorithm as being equivalent to a Turing machine program, as in the Church-Turing Thesis, is useful for formal studies of computational theory. Other formal models of computation can be substituted, since the thesis essentially claims that all sufficiently rich models are computationally equivalent; that is, they can compute any function that a Turing machine can and vice versa. To be complete one needs to prove this equivalence. For example, one might be able to prove that BeeperBot with an infinite grid, and PET with unboundedly large arrays are each computationally equivalent to Turing Machines. But this definition is not very useful for our purposes.

Why do we "sort of like" the last definition? Because it indicates what we will look for when we ask you for an algorithm. Let us break down the definition to explain each of its parts.

executable statements *Statements* can be thought of as statements similar to a PET program, and *executable* means that we can easily translate them into PET code that will execute.

finite set This just means that we cannot keep on adding statements to the program for ever. We must have a bounded size program that implements our algorithm.

Ordered This means that there is a default order in which the statements should be executed, and of course we are again thinking of PET as the executor.

defines a terminating process The execution of a program is called a *process*. Most of the time we want a process to do some computation and give us a result. This means it should eventually halt, or terminate after producing an output.

solves some problem In some sense, any PET program that halts is an algorithm that does something. But usually you will be given a specific task to solve, and so a correct algorithm should solve all instances of the problem.

Why do we not "fully like" this definition? Things like operating systems are programs we want to keep running, and not terminate. Often we abuse the term algorithm to include such programs. Also this definition does not easily cover certain classes of algorithms, such as parallel and nondeterministic algorithms. But for our purposes these will not matter, since we do not cover these in this course.

5.1.1 Working definition of Algorithm for CMPUT 101

Do not worry too much about the "formal definitions". If we are using pseudo-code, a term used to mean informal algorithm description, then there is little point in fixating on a formal definition of algorithm. Here is what you need to keep in mind:

> *For purposes of this course, an* **algorithm** *will be a set of steps that we can easily implement in a PET program that solves some problem.*

When we ask you to implement an algorithm we mean to write it using PET and test it for correct operation. You should include meaningful and helpful comments, to aid the TA in understanding your code. Uncommented and unclear code will lose marks.

We will also use the term when solving problems using BeeperBot, or other programming systems.

Generality

A Problem is most often not a single instance, but a set of instances. So when we say "solves some problem" we mean it should solve instances of a certain type, not just one instance.

For example, when we discuss sorting, we don't want an algorithm that sorts one specific sequence, instead we want one which works on any array of any size.

This is very important. Answers which only work on one input will typically result in reduced grades.

5.1.2 Some Examples of Non-Algorithms

Yes, of course there are infinitely many things that are not algorithms. So to keep the nit pickers at bay, let us explain that in this section we look at ideas that might be thought to be algorithms but are not.

We find that people often present things similar to these as algorithms, without realizing that their description is too vague to be implemented.

A non-algorithm to detect primes

Recall that a *prime number* is an integer bigger than one whose only divisors are itself and 1. Some primes are $2, 3, 5, 7, 11, 13, 17, \ldots$ and the list goes on forever. We would like an algorithm that will take as input a positive integer, and indicate whether or not it is prime. Here is an approach

```
int p
define main {
    read p
    if ( p == 1) {
        writeline "no"
```

```
} else {
    if ( p == 2) {
        writeline “yes”
    } else {
        if ( p == 3) {
            writeline “yes”
. . . and so on and so on . . .
```

Since there are infinitely many primes this program is not finite, and so can never be completed. It is not an algorithm.

Note if we only wanted to determine primality of integers less than some fixed bound, say 1000, then we could write an algorithm in this way. The issue here is that to be completely general, if we continue in this fashion we have to write a never ending sequence of **if** statements, and that violates our condition that there must be a finite set of statements.

Twenty Questions Game

There are many variations, but generally this two player game starts with one player thinking of an item, and the other asking questions which can be answered “yes” or “no”. For example the first question might be “Is it alive?” This is sufficient to divide the universe into two parts, living things and non-living things. For each of these outcomes, there will be another question dividing the result and so on.

This concept is illustrated in figure 5.1.

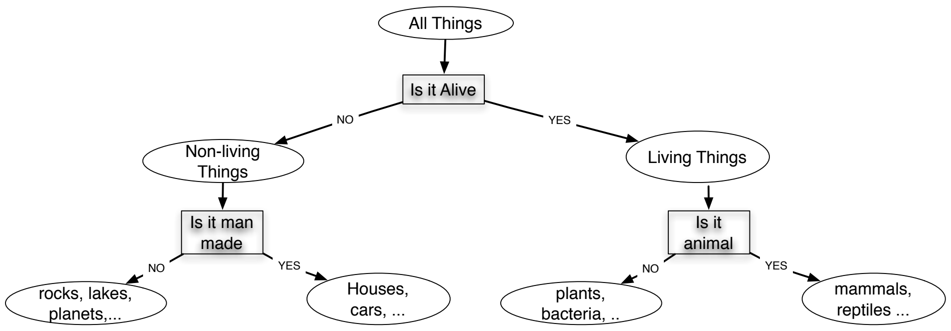


Figure 5.1: The first two levels of a Twenty Questions Decision Tree

From the figure, we see that after two questions we have split the original universe into 4 parts, and know our answer is in one of them. The diagram is commonly called a *decision tree.* The top node we call the *root* and the bottom nodes are the *leaves.* Computer scientists are notorious for their upside down trees.

Each time we allow another level in the tree we double the number of possible outcomes. So for k levels, we would have $2 \times 2 \times 2 \ldots \times 2$ k times, or 2^k possible items. With twenty levels, meaning that the player gets to ask twenty questions in a row, the player should be able to distinguish between 2^{20} or about 1,000,000 different answers. However, if you start to implement this, you will quickly see that you need to think up 1 million questions in advance. And if you really allow all the infinity of things that might be in the universe, then you will need an infinite program.

So what is wrong? Generally, when people play this we do not start with a fixed set of questions, but instead make them up as we go. We make assumptions about what the host is thinking about, get hints from the host's reactions, and take into account other clues such as what we both can see at the time, and results of previous games.

Despite much research effort, at this time we simply do not know how to program a computer to do what children can do, that is make up reasonable questions as we go along. Artificial intelligence lags far behind human capability so far.

However, when we have a finite size universe, or have a simple way to make up the questions, then this decision tree approach can be very powerful indeed, as we shall see in the following sections.

Cooking Recipes

A favorite algorithm example of many text books is a cooking recipe. However, how do you write a program to "salt to taste"? Recipes in fact are full of vague instructions that must be interpreted by the judgement of a human, and for which we know of no reasonable way to write a program.

It may be possible to design algorithmic recipes that could be executed using a robot in a setting where all the ingredients and the facilities layout are precisely specified, but the general run of the mill recipe from your grandma's handwritten collection, cooked in an average home kitchen is currently beyond our algorithmic capabilities.

5.1.3 Some Elementary Algorithms

An algorithm can be specified in many different ways or levels. For example, it can be described in a natural language.

Celsius to Fahrenheit

Consider converting temperature from Centigrade to Fahrenheit. One way is to say it in English:

> Multiply Centigrade by 9 divided by 5 and then add 32 to the result.
> Then you get the degree in Fahrenheit.

Because the conversion can be specified more mathematically, it can be specified as:

$$F = C \times 9/5 + 32$$

In the context of PET programming, this algorithm can also be specified this way

```
int F, C
define main {
  writeline "Enter Degrees C"
  read C
  F = ( C* 9)/5 + 32
  writeline "Degrees F to nearest degree " F
}
```

Finite Decision Trees

It is also possible to specify a decision tree algorithm in PET if you have a finite set of items. Let us say you have to guess from the set of items { ret hat, red coat, blue shirt, blue tie }. Clearly the first question should distinguish the color of the object. Here is a program whose code can be found in the file *decision.txt* in the sample programs.

```
# decision tree program to distinguish between
# red hat, red coat, blue shirt, blue tie
# user must respond with yes or no
string answer
define main {
  writeline "Is it red?"
  read answer
  if (answer == "yes") {
      writeline "Is it a hat?"
      read answer
      if ( answer == "yes" ) {
          writeline "It is a red hat"
      } else {
          writeline "It is a red coat"
      }
  }  else {  # not red, must be blue
      writeline "Is it a shirt?"
      read answer
      if ( answer == "yes" ) {
          writeline "It is a blue shirt"
      } else {
          writeline "It is a blue tie"
      }
   }
}
```

Note that there are three questions in the program, but in any one run, only two will be asked. If we had 2^k items, and could divide them evenly on each question, we would need $2^k - 1$ questions in the program, but only k of them would be asked in any one run.

Nevertheless, the program would be tedious to write, especially if say $k = 20$.

Hint on Assignment One Question 1.

The first question on the assignment asks you to make a decision tree algorithm and implement it in BeeperBot. This algorithm must distinguish which of 6 letters is sitting above the robot. You will have to have at least 5 questions of the type "next_to_a_beeper" in the program. But, if you ask questions cleverly, you should not have to ask more than about three questions in any one run. How well can you do?

5.2 Conversion Algorithms

We have seen in previous chapters that numbers in computers are stored in a binary format. It is with some irony then that we now undertake to write programs to convert numbers and print them in binary format, and to take sequences of bits and print the base ten form of the number it represents.

The problem is that PET hides the details in the lower levels of implementation from the user to a large extent. In fact, one of the goals of modern languages is to provide better abstraction, and this means that more of the low level implementation detail is hidden. But computing scientists, and also mere programers, should have an understanding of what is underneath. Otherwise how are they to understand behavior such as that in the questions in section 4.4?

This section will outline part of the programs, but leave part as exercises for the student to complete.

5.2.1 Base Ten to Binary

PET normally takes integer input as a sequence of the digits $0, \dots, 9$ representing the number in base ten. In this section we will take such a number and output it as a standard binary number. Well, actually we will do part of that and leave the remainder as exercises.

Recall the method of converting to binary; we repeatedly divide by 2 making the remainder the next bit. This is the method expressed as an algorithm, modified to always compute a 32 bit representation.

```
input the number $b$
i = 0
while ($i < 32$ ) {
    bit[i] = remainder of b divided by 2
    b = b/2
    i = i + 1
}
```

A few things need to be noted. First, we are storing the bits in an array called bit. We will need to declare this to have size 32.

Second there is the statement "remainder of b divided by 2". This does not look like a PET statement, so is this really an algorithm?

Well, it is if we can turn this into PET code. Here is how we do this, using a subroutine. If you forget your grade 3 arithmetic, see the file *Links.html.* First we present the subroutine we call **quotient_and_remainder**.

```
# here are the variables needed in
# the subroutine quotient_and_remainder.
int dividend      # must be set by main program before each call
int divisor       # must be set by main program before each call
int remainder     # set by the subroutine
int quotient      # set by the subroutine
int temp          # for internal subroutine use

define quotient_and_remainder {
   quotient = dividend / divisor
   temp = quotient * divisor
   remainder = dividend - temp
}
```

Notice that this subroutine requires several variables. These variables can be divided into three classes to help us distinguish their role.

Arguments As used, the variable **dividend** is an argument to this subroutine. If we wanted to use quotient and remainder for other divisors, then **divisor** would also be an argument. In general, the arguments are the variables that act as *inputs* to the subroutine.

Results The results or *return values* as they are often called, are the values the subroutine computes that the calling program needs. In this case, we want the **quotient** and **remainder**.

Internal The internal variables are ones the subroutine needs to do its job, but are not useful outside the subroutine. This subroutine uses the variable **temp**. As used in the program below, we might also think of **divisor** as internal, since it is never changed by the main program. But in general usage, **divisor** should be an argument.

Now to use this subroutine we must assign values to the arguments before each call, and copy the results where we need them after the call. With this subroutine, we can write the above algorithm using PET as follows.

```
  i = 0
   while ( i < N ) {
       dividend = copy              # need to provide the dividend
       divisor = 2                  # and divisor
```

```
        call quotient_and_remainder  # do the division by 2
        copy = quotient              # copy is now copy/2
        bit[i] = remainder
        i = i + 1                    # ready for next bit
    }
```

Of course we need to declare the array **bit** and other variables. Here is how we declare the array.

```
int N = 32
array bit[N]
```

A program with all these pieces can be found in the file *convert_ten_to_binary.txt* in the PET examples. Note that the value **N** is set to 32, the number of bits we need. **copy** is a copy the program makes of the input value, so we can re-use the input value later. Notice that **copy** will be reduced to 0.

The exercises ask you to complete the program and modify it it in various ways. You should complete those exercises relating to the program in the file *convert_ten_to_binary* before proceeding.

5.2.2 Binary to Base Ten

If we have a number stored as an array of bits as in the previous subsection, then we can write an algorithm to perform the standard binary to base ten conversion for earlier chapters as follows. First, we need to keep a value for the number we are computing. We calls this **result**. Then we need to remember to compute a power of two as the amount to be multiplied by the next digit. For this purpose we have a variable **multiplier** which we initialize to 1. Each time we move to the next digit, we multiply this by 2. Here is the algorithm in PET code.

```
    multiplier = 1
    i = 0
    result = 0
    while (i < N) {
        result = result + bit[i] * multiplier
        multiplier = multiplier  * 2
        i = i + 1
    }
```

Now we only need to input the binary number. Unfortunately, PET does not make this easy for us. We would like to simply type a string such as “1101”, but PET does not provide any way to parse a string to see what the individual characters are.

So, we are forced to input the bits one digit at a time. If you try this you will quickly see that you do not want to enter all the leading zeroes to fill in all 32 bits for numbers such 1101. So, we need to provide a way to stop the input part of the process as soon as we have entered all the relevant digits. To accomplish this, we allow the user to enter a number other than 0 or 1 to act as an "end of input" flag.

Here is the code to accomplish this. Note that this requires the user to enter the digits from the right to the left, that is starting with the least significant.

```
i = 0
stop_flag = false
while ((not stop_flag) and ( i < N) )  {
   writeline "enter bit " i
   read bit[i]
   if ( (bit[i] < 0 ) or (bit[i] > 1) ) {
         stop_flag = true
         bit[i] = 0 # fix the bad bit
   }
   i = i + 1   # ready for next bit
}
```

5.3 Searching Algorithms

We will take the approach of solving particular tasks, and then use the solutions to identify certain types of algorithms such as sequential or binary search.

5.3.1 Guess a Number Game

Our first problem is based on a much simplified version of the "twenty questions" game. In this game, the *host* thinks of a number in the range $1 \ldots N$, where N is agreed on in advance. For most of our examples we will let N be 100. The *player* must guess what the number is.

We will write, or discuss a variety of algorithms based on this problem, programming both the host and the player as separate PET programs, which should

As with the twenty questions game, the player is only allowed to ask restricted questions. For example, the player cannot ask a question such as "what is the last digit of your number when written in base ten?", because the answer could be any of $0, 1, 2, 3, \ldots, 9$. When $N = 100$, two such questions would always determine the number! You might as well allow the question "what is the number?" which does not make for a very interesting game.

We might modify the previous question by asking "Is the last digit = 9, when written in base ten?" This question can always be answered yes or no, and so is a Boolean question, but we still have a problem. If we allow such questions, we have no way to program the host using PET, because PET does not provide any way to parse strings of words. But even if we had access to a powerful modern programming language, allowing for arbitrary questions of this sort would be very difficult. The reason is that this requires a level of understanding of natural language that is far beyond this course, and in fact no program even exists at the level of human intelligence.

So, we will restrict our game to allowing only a predefined type of question. In all cases, the user[1] will only ever be asked to enter either a number, or one of the words"yes" or "no" or a boolean value of " true" or "false", or one of the three answers "Hi", "Low" , or "Correct".

5.3.2 Guess a Number Host

We will first build a the host program. The first difficulty we encounter is how to generate a random hidden value. Well, on PET nothing is hidden, so we will have to pretend that the memory display is not visible.

We still have the issue of generating a number at random. If the game is to be played with the hidden number between 1 and N, then we need to generate such a number. Most systems provide some kind of pseudo-random number generator, which is subroutine that is built in to the language. Starting with an integer, it generates a new integer each time it is called, and the sequence looks random.

PET does not provide an explicit generator, but it does have the command **fillarray**. Given an array of size N, it fills the array with random values in the range $[1 \ldots 10 \times N]$. For example, the following code when executed creates an array of size 10, and fills it randomly with values between 1 and 100.

```
int hidden         # the hidden value generated by this routine
int N = 10         # hidden will be between 1 and N*10
array A[N]         # use fillarray and then take the 0th element
int limit = N*10   # upper limit to the range of guessing

define generate_hidden {
     fillarray A
     hidden = A[0]
     writeline "My hidden number is between 1 and " limit
}
```

The original purpose of this command was for use in creating arrays to illustrate sorting algorithms, and we will use it for that purpose later. Note that we use `A[0]` as the hidden number.

When building a program, it is a good idea to abstract out components as subroutines as illustrated here. Now that we have this subroutine, it makes sense to test it. We will need to declare a variable `int i` for the following.

```
define main {
    # test the hidden number by calling it ten times
    i = 0
    while ( i < 10 ) {
```

[1]We refer to the person running a program as *the user* whether she is host or player.

```
        call generate_hidden
        # print the hidden so the user can verify
        writeline "      Hidden = " hidden
        i = i + 1
    }
}
```

The complete test can be found in one of the on line codes. It may differ slightly from that developed in class.

Sequential Search Support

When we search for something by simply trying every possibility in some order, then we call it *Sequential Search.* For our game, the following would be different sequential searches because they imply different orders. But in each of them, in the worst case scenario, we would have to check all N different numbers.

- Increasing: "Is it 1?", "is it 2?", " is it 3?", ...
- Decreasing: "Is it 100?", "is it 99?" ...
- 'Even, then Odd: "Is it 2?", "is it 4?", ... "is it 1?","is it 3?" ...

In the following we have a subroutine to play one game, where the user types a number and the game will only respond correct or wrong. We say this *supports* sequential search, because sequential search is the only realistic way to play this game (assuming we could not cheat and look at the hidden variable value in memory). We have deleted the routine **generate_hidden** which appears above.

```
# GUESS A NUMBER SEQUENTIAL HOST

bool play_again      # used to control multiple games
string response      # to hold players reply to play again?

bool found           # used in the routine play_one_game
                     # to indicate when hidden has been found
int guess            # to take users guesses as input

# routine to play one game
define play_one_game {

   call generate_hidden     # start by hiding a number

   found = false            # the player has not found it yet
   while ( not found) {     # keep searching until found

       writeline "Enter your guess"
       read guess
```

```
        write "Your guess of "
        write guess

        if (not ( guess == hidden) ) {  # still not it
           writeline " is wrong."
        } else {
           writeline " is CORRECT!"
           found = true               # this will end the search loop
        }
     }
}


define main {
  play_again = true         # assume they want to play once at least
  while (play_again) {
     call play_one_game

     writeline "Play again(yes/no)?"
     read response

     # if response is anything but "yes"
     # play_again will be false, and the game will stop.
     play_again = (response == "yes")

  }

}
```

The complete program can be found in the set of programs for this chapter.

Binary Search Support

Doing a sequential search takes a very long time. The game can be made to be more interesting if the program provides a bit more information. Here is the idea. Instead of just "correct" or "wrong", the program will answer "Too high" if the guess is bigger than the hidden value, "Too low" if the guess is lower than the hidden value, and of course "Correct" when it is correct. Note that this is how the game is played on TV game shows.

Implementation of this is left as an exercise. You only need to modify the subroutine **play_one_game**, making the appropriate tests, and giving the needed response. Note that you will have to insert another **if** statement.

We say this now *supports binary search.* What do we mean by binary search?

Well, the player using this host has significant advantage over one using the version given above. Suppose as usual that $N = 100$. For the first guess, the player can choose the number that splits the set of possible answers in half. So, for the first guess, choose guess equal to 50. Now if the response is “Too high” then the player knows the answer is in the range 1 and 49. If the response is “Too low” then the player knows the answer is in the range 51 and 100. (And of course if “Correct” then the game is done).

In either of the first two cases, with one guess, the number of values that the hidden number could be is reduced by about 1/2, from 100 to 49.

Now suppose the player is smart. She creates a mental image where she keeps two fingers on a sequence of 100 numbers. After guessing 50, according tot he above knowledge, she can either move her right finger from 100 to 49, or she can move her left finger from 1 to 51. Suppose the second case happens. The next guess should be in the middle again, or $(100 + 51)/2$ which after throwing away the fraction is 75. After the answer to this guess is given, then at most 25 number will remain as possibilities, either the hidden value will be known to be in the range $51 \dots 74$ or in the range $76 \dots 100$.

Using this method, each guess cuts the range approximately in 1/2. Let us set up a little table, where we on the left we keep the number of the guess (i.e. k counts the number of guesses, not the number guessed) and on the right *approximately* how many possibilities are left.

k	Remaining
1	50
2	25
3	13
4	7
5	4
6	2
7	1

Note that after 7 guesses, there can be at most 1 number remaining in the range of 100. So, one more guess and we have the answer. Notice that if we multiply $2 \times 2 \times 2 \ldots \times 2$ k times, or 2^k then the value is 128, which is the smallest power of 2 larger than 100. We say that k is the *base 2 logarithm of 100.*[2]

In general, for any N, this binary search procedure will need $k = \log_2(N)$ questions to get down to one possibility for the answer. Or equivalently, k is the smallest integer such that $2^k \geq N$.

So for example, if we set the search range $1 \dots 1024$, we would need at most 10 questions to narrow the set down to one possibility. If we had $N = 1000000$ then we need about 20 questions and so on. Recall the discussion in a previous chapter on the powers of 2.

This is an enormous savings in the number of questions we need.

Of course, the player does not have to do a binary search with this host. She can just ignore the benefits that are available, and do a sequential search. But

[2]To be more mathematically precise $k = \lceil \log_2(100) \rceil$, but we will ignore the ceiling operation for this course.

for $N = 1000000$, this could take a very long time.

In computing science, we say that *binary search* uses $O(\log N)$ (read as Big-Oh of Log N) time to do the search.

5.3.3 Illustration of the Binary Search Decision Tree

In order to use binary search, we need to keep track of the current range where we think the answer might lie, and then find middle as our guessing point. If we think that the range starts at an integer *low* and ends at *high* then we can designate the range by the notation $[low \ldots high]$. We say that *low* is a **lower bound** on the range, while *high* is an **upper bound**. (We use lower_bound and upper_bound as names for the range in the player program.)

Now to do the search, we compute the middle as $middle = \frac{low+high}{2}$. The middle value is our guess.

If the host answers "correct" then we are done. However, if the host answers "too low" or "too high" then we must prepare to guess again. Before doing this we must discover the new extent of the range.

Suppose the host answers "too high". This means the hidden number is less than our current upper bound, which we call here *high*. So, we need to make *high* smaller, in particular we can safely set $high = middle - 1$, since the middle is known not to be correct. Alternatively, if the host answers "too low" then we know the number is larger than the *middle* we guessed, so we set $low = middle + 1$.

Notice that in terms of the fingers used earlier, exactly one of the fingers moves after an incorrect guess.

In figure 5.2 we show part of a decision tree that results from this thinking when $N = 10$. Compare this to figure 5.1. Ignoring the branches labeled "correct" we see that the main difference is that in the guess a number problem we have a very regular way to generate the questions.

Notice that in any one game, the questions and answers will follow down one sequence of branches until a branch labeled "correct' is reached, assuming the player always chooses the middle number as shown. Thus, there is one such 'correct" branch (called a *leaf*) for every number in the range. We should note that if the algorithm is done correctly, and the host never lies nor cheats, then leaves labeled "error" should never be reached. For example, at the bottom left of the tree, since the range is $[1 \ldots 1]$ the answer must be 1, so when the guess 1 is made it must be correct. So, the if the host answers "too high" or "too low" it must be an error.

The fundamental reason we have this method of generating questions for this decision tree is that the numbers are ordered. What this means in general this will become more apparent when we discuss binary search of a sorted array in the next chapter.

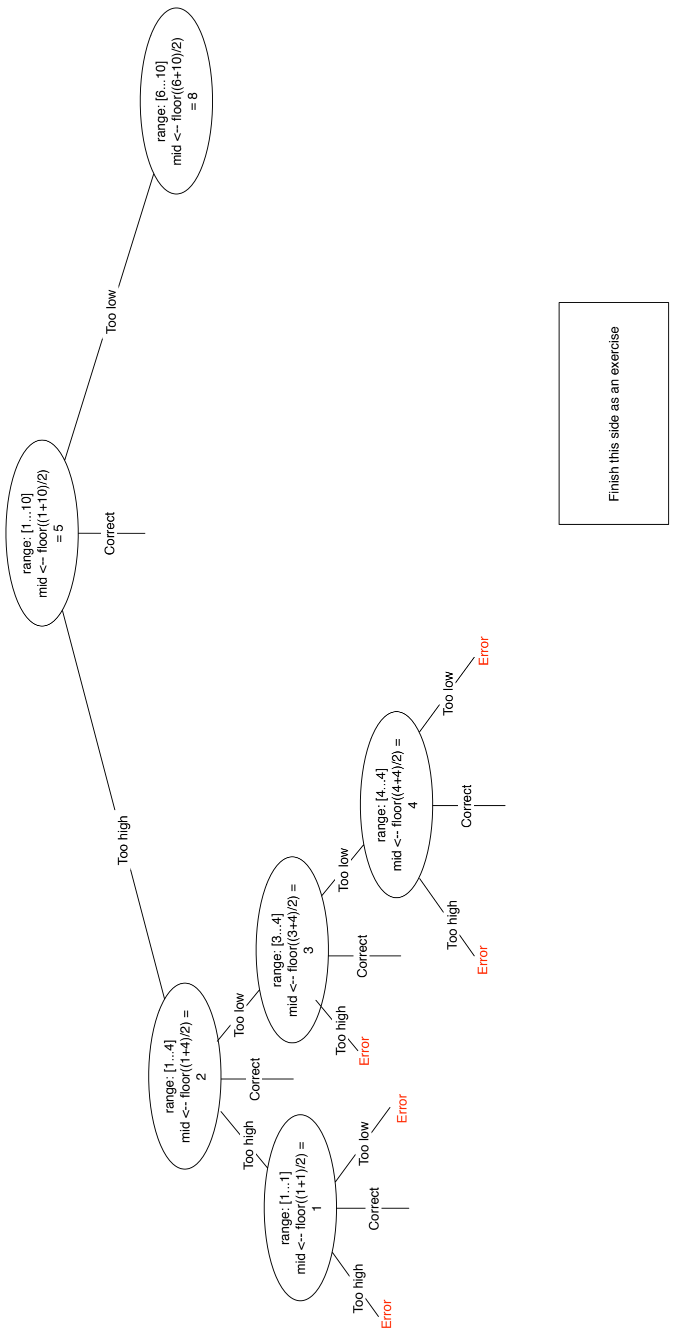


Figure 5.2: Partial Binary Search Decision Tree for $N = 10$.

5.3.4 Player for Guess a Number: Binary Search

It is quite easy to program a player to do sequential search. You just start the program guessing at `guess = 1` and keep looping, adding 1 to the guess each iteration, and stopping when the host says "correct". It is also boring to use. We leave this as an exercise.

Programming the player to do a binary search on the other hand is more interesting, and requires more understanding of the binary search algorithm. The following code is in the file *binary_player.txt*.

```
int N = 100
int lower_bound
int upper_bound
int middle
bool found
string answer

define main {
     found = false
     lower_bound = 1
     upper_bound = N
     while ( not (found) ) {
        middle = ( upper_bound + lower_bound) / 2
        writeline "My guess is " middle
        writeline "Is it high, low or correct"
        read answer

        if ( answer == "high") {
              upper_bound = middle - 1
        } else {
              if ( answer == "low" ) {
                   lower_bound = middle + 1
              } else {
                   if ( answer == "correct" ) {
                         found = true
                   } else {
                      writeline "Unknown answer"
                   }
              }
        }
    }
}
```

In the code above, **lower_bound** and **upper_bound** act as described previously. These delineate the current range where the hidden number might be. Initially we set these to 1 and N respectively, as that is the full range allowed by the game.

To do binary search, the game simply guesses the integer at the middle of the range. This is computed by adding the two numbers together, and dividing by 2. If the host says the guess is too high, then we know the upper bound can be changed to the middle minus 1. It could not be the middle, because the user should have answered "correct" if that were the case. It cannot be bigger than the middle, because the middle itself is too high.

Similarly, if the answer is "low" then the lower bound is set to the middle plus 1.

There are some exercises to try. What happens if the user lies at some point? For example, what if the user just replies "low" all the time?

One of the exercises asks you to modify this program to handle such bad behavior.

5.4 Algorithm Questions

We separate the questions in this chapter by section, but be aware that questions for later sections assume you also know the material in earlier sections. Starred questions are for those wishing a little more challenge, and deeper insight into algorithmic issues. They are not indicative of questions you would be asked on an exam in this course.

5.4.1 From section 5.1

1. Draw the decision tree that represents the hat/coat/shirt/tie program in section 5.1.3.

2. Write a PET program to convert from Fahrenheit to Centigrade.

5.4.2 From section 5.2

1. **Print:** Write an output subroutine to complete the program to convert base ten to binary in the file *convert_ten_to_binary.txt.* To do this, write a subroutine that outputs the 32 bit binary number that is stored in the array **bit** on one line. This should print the binary number in the correct format, with the highest order bit on the left. That is, if the user entered the number 13, the output should be

   00000000000000000000000000001101

   and **NOT** 10110000000000000000000000000000.

   Note: you may allow spaces between the characters, since PET is a bit idiosyncratic on this issue. However, if you prefer to output without spaces, then a hint is that the statement

   ```
   write "1"
   ```

will print the “1” without prepending a space.

2. **Better Print:** Modify the output subroutine in the previous question so that it suppresses leading 0’s. Thus, to print the binary representation of 13 you should only print 1101. Note: if the input number is zero, then a single digit “0” should be printed.

3. **Efficiency:** If you run the program in the file *convert_ten_to_binary.txt* and input value 13, you will notice that most of the time is spent running through the loop setting leading bits to 0, which is wasteful since the array is already initialized to 0. Modify the program so that once the value in copy is 0, the program quits the **while** loop.

4. **Generality:** Modify one of your programs based on the program in the file *convert_ten_to_binary.txt* so that it can convert the input number to any positive base $1 < b \leq 10$. You should ask for b as an input. Start with a version that has a suitable output routine from the previous questions. **Representation**

   (a) What happens if you try to use base $b = 1$?

   (b) What is the issue if you try to use a base $b > 10$?

5. **General Conversion to Base Ten:** Modify the program to convert binary to base ten in the file *convert_binary_to_ten.txt* to ask for an input value b with $0 < b \leq 10$ and then treat the rest of the input as a number in base b. This number should then be converted to base ten and output.

6. Note that if you do the previous two questions in this section you will be able to convert from any base $0 < b \leq 10$ to any other. Combine the two relevant programs into one program.

7. For the unmodified program in the file *convert_binary_to_ten.txt* what happens if you enter all 32 binary digits and enter a 1 for bit[31]?

8. Write an algorithm to convert a 32 bit positive integer in binary to its negative value in the two’s complement representation. Hint: start by computing the one’s complement.

9. Write a PET program to read a positive integer n and then print out each divisor. Recall a positive integer p is a divisor of n if it divides it with no remainder.

10. Write a PET program to read a positive integer n and determine if it is a prime number. Recall that n is prime if $n > 1$ and its only divisors are 1 and itself. Hint: Consider either the previous question or the Sieve of Eratosthenes in one of the questions in section 4.4. These will lead to different algorithms, one using an array. What is a possible issue with using an array for this question?

5.4.3 From section 5.3

1. Consider the program in the file *sequential_host.txt*. A *session* is any one run of the program, which may involve playing several games. Make modified programs to do the following.
   (a) Count the number of guesses the player uses, and report this when each game is finished.
   (b) After each game, report the number of guesses used in that game, and also the average number used in all games played so far in this session. The average is defined as (total guesses over all games) / (number of games played).
   (c) Place an upper limit on the number of guesses allowed for any one game, and if this is exceeded without finding the hidden number, then that game stops and is reported as a loss by the player. Keep a record of wins and losses and report this after each game, and after the user stops playing.
2. Starting with the program in the file *sequential_host.txt*, make a host that supports binary search as discussed in class. That is, the host should report "high", "low" or "correct" for each guess. Then modify this program to add all the enhancements of the previous question.
3. Write the player program for the sequential search approach. Modify it to report how many guesses are used.
4. Complete the binary search decision tree in figure 5.2 for $N = 10$.
5. Modify the program in the file *binary_player.txt* to detect cheating on the part of the host, as mentioned in the text.
6. One variation of the "Guess a Number" game is the "Unbounded number guessing game". In this the host is allowed to choose any positive number and asks you to guess it. Solving this efficiently requires a different approach. To get an $O(\log n)$ search, you can start with a small guess, then keep doubling your guess until an upper bound is found. After that you use binary search. Modify the binary search player program for "Guess a number" to play the unbounded version.
7. **** Creating an Adversary.** The idea here is to create a host that *kind of* cheats. You should write a host program supporting only sequential search that will always make the player use the maximum number N of guesses. The cheat is that the program does not actually think of a hidden number, it simply keeps track of the guesses made, and keeps saying "no" until only one unguessed number remains. Then it says "yes" when that last number is guessed. NOTE: you **cannot** assume that the player always guesses in the order $1, 2, 3, \ldots N$. Also, your answers must be *consistent*; that is, if you say "no" to a guess of 13, and then later the player again guesses 13 you cannot then say "yes". Hint: use an array.

8. ****More Adversary.** Similar to the previous question, create an adversary host that supports binary search. The idea is that if the player does not choose the middle of the current *[low . . . high]* range then the adversary picks the answer that leaves the largest possible subrange to contain the hidden value. Be careful, this also must be consistent, and that is trickier here. For example, you cannot assume that the player will even pick a number in the current known range. You must remain consistent under any sequence of guesses the player might decide to use, even guesses that are negative or larger than the upper limit N.

Chapter 6

Introductory Searching and Sorting on Arrays

For many problems it is convenient to have collections of data, that is variables, that are associated together under one name. In PET, one type of collection is provided, namely the **array** discussed initially in Chapter 4 and used a bit in Chapter 5.

In this chapter we discuss searching and sorting algorithms for arrays. As part of this, we want to illustrate some of the properties of algorithms we are interested in, principally efficiency, or the number of steps the program uses. As arrays, or problems in general, get larger the differences in how many steps different algorithms take to do the same task can be enormous.

As with most introductory courses, we focus mainly on simple mechanics of the algorithms, and our analysis is very simple and informal.

6.1 Sequential Traversal of an Array

If you need to, revisit chapter 4 and the declaration and descriptions of arrays. Here is the basic form of most array traversals, which we refer to as a sequential traversal.

```
int N = 25
array A[N]
int i

define main {
  fillarray A
  i = 0
  while ( i < N){
      # do something with value A[i]
      i = i + 1 # next element
  }
}
```

We precede the actual traversal with a **fillarray A** command, which when executed fills the cells of the array with integers randomly chosen in the range $[1, 250]$, or in general $[1, 10 \times N]$.

We use this form, suitably modified, throughout this section.

6.1.1 Addition and Other Functions on an Array

We use the basic form to compute the average of a randomly filled array. The average is defined as the sum of the values divided by the number of values. Here is a program to do this.

```
# CMPUT the average
int N = 25
array A[N]

int i

int sum, avg

define main {
  fillarray A

  sum = 0
  i = 0
  while ( i < N){
      sum = sum + A[i]

      i = i + 1 # next element
  }
  avg = sum / N
  writeline "Average is " avg

}
```

Recall the quotient and remainder function from chapter 5. Here we use this to illustrate that using arrays we can do a range of functions that work over pairs of arrays. The following, for each i, divides the larger of two corresponding values by the smaller and stores the remainder in a third array. Its only purpose is to illustrate a slightly more complex array example.

```
# Fill C with the remainder of larger divided by smaller
int N = 10
array A[N], B[N], C[N]
```

```
int i

int dividend      # must be set by main program before each call
int divisor       # must be set by main program before each call
int remainder     # set by the subroutine
int quotient      # set by the subroutine
int temp          # for internal subroutine use

define quotient_and_remainder {
   quotient = dividend / divisor
   temp = quotient * divisor
   remainder = dividend - temp
}


define main {
  fillarray A
  fillarray B

  i = 0
  while ( i < N){
      if ( A[i] > B[i] ) {
            dividend = A[i]
            divisor = B[i]
      } else {
            dividend = B[i]
            divisor = A[i]
      }
      call quotient_and_remainder
      C[i] = remainder

      i = i + 1 # next element
  }


}
```

There are a number of exercises in the questions at the end of the chapter that ask you to compute various functions over the elements of an array.

6.1.2 Searching for an Element

Here is the basic algorithm done in class for sequentially searching an array for a given value. This program can be found in the file *seqfind.txt*.

```
# Sequential array search
```

```
int N = 15
array A[N]

int i
int loc
# we will write a funciton to sequentially search
# for the following value in the array
int search_value

define main {
  fillarray A

  writeline "Enter numer to search for"
  read search_value

  loc = - 1
  i = 0
  while ( i < N){
      if ( A[i] == search_value ) {
         loc = i
      }
      i = i + 1 # next element
  }
  if (loc < 0 ) {
     writeline search_value " value not found"
  } else {
     writeline "The  value " search_value " is in location " loc
  }
}
```

Note since PET displays the internal memory, including the array A after it is filled, it is easy to test this program. First you should look for values in the array. Then you should look for values not in the array. You may want to also test that it finds the value if it is in the 0th or $(N-1)$th locations. Checking these end points of the array is known as checking the boundary conditions. This particular program is fairly simple, but in more complex programs such cases are often the sources of errors.

6.1.3 Finding the Maximum in an Array

Now we want to find the maximum element in an array. This differs slightly from searching for a given value, since we do not know either what value the maximum will be or where it is. We need to keep track of the biggest we have seen so far in the traversal of the array. If we see a bigger value, then we switch to that one as the biggest so far. But where do we start? Well, if we could assume the array values are all positive, we could start by setting the known maximum to a negative value. But this is dangerous, because we might want to do a search sometime on an array whose values are all negative.

It is better to start without making any assumption about how big or small the values might be. Suppose that we start at index 0. Then the biggest we have seen *so far* is $A[0]$ in location 0. So, we initialize our *maximum* and *maximum location* to $A[0]$ and 0 respectively, and then iterate through the rest of the array checking to see if there is a bigger value we should use. Here is the code which can be found in the file *findmax.txt*.

```
# Sequential array search to find the maximum
int N = 25
array A[N]


int i
int loc, max

define main {
  fillarray A

  # we start by setting the maximum
  # to be the first element
  max = A[0]
  loc = 0

  i = 1
  while ( i < N){
      if ( A[i] > max ) {
         loc = i
         max = A[i]
      }
      i = i + 1 # next element
  }
  write "The maximum is in location " loc
  writeline " with value " max

}
```

See the exercises for some variations that you should work out.

6.1.4 Run Time Stats: Find Max

We now consider one of the properties we strive for in algorithm design, namely making an algorithm that executes as quickly as possible. In general, we want an algorithm that is competitive against other algorithms as the size of the instances increases.

We will look at the algorithm for finding the maximum in terms of its **Run Time Stats** as recorded in the lower right pane of the PET tool. We remind the reader that the code we are considering is presented in the subsection 6.1.3, and can be found in the file *findmax.txt* in the sample programs for this chapter.

The line labeled **All** in the **Run Time Stats** pane counts the total number of statements executed by the program. If you run the program slowly, or step through the program line by line, you can see this counting in action, as each step increases the count by one.

In addition to the total count, you can obtain the count for any individual line of the program. To do this, select the line of interest, and right click (or control click if using a single button mouse), then in the pop-up select **Toggle Statistics Line**. This will cause a green highlight to appear on the selected line.

We suggest you try this now on the line "`if ( A[i] > max ) {`", which is line number 19 in the original program. (We will refer to this as line 19, although it may be on a different line number if the program has been edited since these notes were written. If this is the case, then select the line that has this text, and substitute your line number for 19 in what follows. A similar comment applies to all other line numbers.)

If you now run the program you will find that in the **Run Time Stats** pane a new line entry occurs, which displays the count of the number of times this particular line is executed. To stop the stats for a particular line just reselect the line and toggle again. You may have stats turned on for several lines at once.

Now we want to determine how this program behaves as we change the size of the array. But first there is a problem. If you run the program several times on the same size array, you will find that the number in the line **All** differs from run to run.

The reason is that the lines of the **if code block**

```
        loc = i
        max = A[i]
    }
```

which are lines 20, 21 and 22 in the original code, will execute a different number of times depending on the contents of the array A. Since "`fillarray A`" in line 10 creates a different set of numbers for each run, the totals vary. To see this, toggle the statistics on for line 20, and run the program a few times, watching the results for line 20. If you still do not understand why this variation happens, you should change the array size to 5, and step through the program a few times line by line.

On the other hand, with $N = 25$ line 19 always executes 24 times. This is because we always do the comparison of **max** to every element but the first, so we always do $N - 1$ comparisons. Note that each of the lines 20, 21 and 22 can only execute **at most** $N - 1$ times, since they only execute if the **if** condition is **true**.

You can also verify that line 18, the **while** statement, always executes N times, and lines 23 and 24 each execute exactly $N-1$ times. Adding these all up we see that the maximum total for lines 18,19,20,21,22,23 and 24 is **at most** $6 \times (N-1) + N = 7N - 6$. All other lines execute exactly once, for an additional total of 12. You may want to check this. Remember, don't count blank or comment lines.

So, in the *worst case* the program executes about $7N$ statements.

On the other hand, in the best case, the lines 20,21 and 22 never execute. This saves about $3N$ from the total, so in the best case the program executes about $4N$ statements. In summary, on an array of size N, the program will execute between $4N$ and $7N$ statements.

At this point you should check the exercises on counting the number of statements executed in this program. These will illustrate what needs to happen to achieve the best and worst case executions.

Note that the *worst case* only uses a constant times N executions.

Computing scientists say that if a program only executes a constant times N statements, then the program runs in *order N time* and use the shorthand $O(N)$ time. Note that the comparisons done in the line 19, "`if ( A[i] > max ) {`" are a good *indicator* of the running time of this program, because we always do N (minus 1) of them.

**Average Case and Harmonic Numbers

In class I tossed out a throw away line that the average number of times the block of code in the **if** statement gets executed is related to the harmonic number H_N. For the curious, here is a brief explanation.

H_N is a function of the positive integer N and is defined as

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$$

Now consider the first time the **if** statement is executed. The current max is $A[0]$, and i is 1. Thus, we execute the **if code block** only if the second element is the biggest of the first two. Assuming that the numbers are distributed "randomly" (uniformly) this should happen 1 time in two, or with the expectation 1/2. Similarly, on the next iteration, the **if code block** will execute only if the third value is the biggest of the first three, so on average 1 time in three. The next iteration execution occurs 1 time in 4, and so on, so in total the average number of times the **if code block** is executed is

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{N} = H_N - 1$$

(The minus 1 is because we do no comparison on element $A[0]$.)

It is known that $H_N \approx \ln N$, which is the natural logarithm of N. (More precise formulae and analyses are available on various web sites, including Wikipedia.) So for an array with size $N = 1000$, we see that the average number of times the **if code block** executes is approximately $\ln(1000) - 1 \approx 5.9$.

We see then that on *average* the **if code block** does not contribute much to the over all running time. But keep in mind that it is possible that the number of times it executes can be anywhere in the range $0 \ldots N-1$, and that the worst case occurs exactly when the array is already sorted in increasing order.

6.1.5 Why not use binary search?

We can ask the binary search question about both searching for an element as in subsection 6.1.2 and about finding the max as in subsection 6.1.3.

When programming a search for an element in an array, it is important to remember that the computer can only "see" one cell at a time. But to use binary search for an item in an array we must be able to ask a simple query (or two), in the form of a location, and determine whether it is the correct location, and if not, we must then know whether the item is to the left or right of the current guess. But the only way to know whether an element is to the left of a particular index when the array is unsorted (random), is to look at all the cells to the left. So, binary search will not work because the items are not in any particular order.

However, we will see in section 6.3 that if the array is **in sorted order**, then we can check a location against the value we are looking for, and by comparing the item to the one we want, we will know whether we should look left, right or already have it.

For finding the max, the problem is even more complex in the unsorted case, because not only do we not know where the largest value is located, we also do not know what the largest value is. However, if the array is sorted in increasing order, then we instantly know that the largest value is in the last position. Thus, for a sorted array, no searching is required for the largest value.

We see that having an array sorted is useful in many problems. Sorting adds information to the set of data that makes it easier to use in many circumstances. But, to have a sorted array, most often we need to sort an unsorted version. This act of sorting is one of the most studied areas of computing science, and so we discuss a few examples in the next section.

6.2 Elementary Sorting Algorithms

In this section we study some simple sorting algorithms in the context of elementary arrays which are initially randomly filled with integers, using the **fillarray** command. As always we declare our arrays using the two line approach

```
int N = 10
array A[N]
```

so that by changing the initial value of N we can try our algorithm on different sizes of arrays.

6.2.1 Swap

One of the first things we will need to do is swap two elements of an array. The thing that trips up many first time attempts is that you must have a temporary storage variable to hold one of the elements, while the other is moved into its place.

We illustrate this in figure 6.1. This figure illustrates the steps required to swap elements $A[i]$ and $A[j]$ where $i = 1$ and $j = 4$. The first step is to copy (i.e. assign) one of the array elements to the variable *temp*. Here we choose to copy the value three form $A[1]$ to *temp*. The second step is to copy the other array element into the first location. The third step is to copy the value in *temp* to $A[j]$.

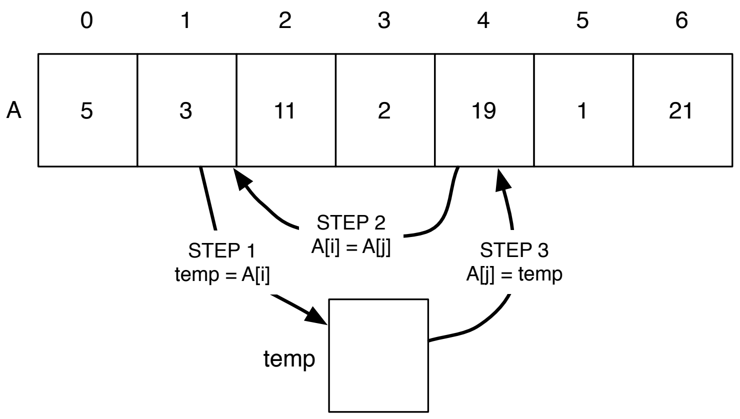


Figure 6.1: Swapping elements $A[i]$ and $A[j]$, where $i = 1$ and $j = 4$.

If we wish, we can define a subroutine **swap** to swap elements to make our program easier to read. We just have to be certain to set the indices correctly before calling the swap routine. Note that a limitation of PET is that we must write a different routine for each array that we might want to perform swaps on. We will do this in the next section.

6.2.2 Selection Sort

Recall that in subsection 6.1.3 we wrote a program to find the maximum element in an array. This technique is also known as *selection*. We can use this idea as the basis for sorting. The idea is quite simple, which we outline here.

```
repeat until sorted
    Find the location of the maximum in the unsorted part
    Swap this with the element at the end of the unsorted part.
```

The idea of the sorted and unsorted parts of the array for selection sort is illustrated in figure 6.2, where the set of values is the same as in figure 6.1, but the algorithm has been running for a few steps.

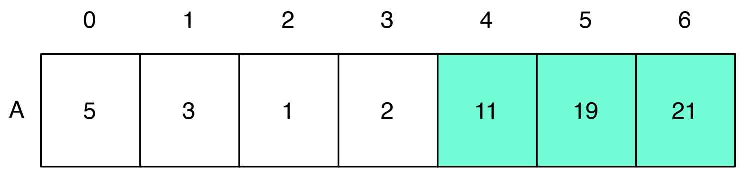


Figure 6.2: Unsorted (left) and sorted (right, colored) parts of an array during selection sort.

At this point the elements 11, 19 and 21 located in positions 4, 5 and 6 are in their proper (sorted) order, and they are in their final locations.

In the next phase, the maximum of the values in positions 0, 1, 2, and 3 will be located. The selection, or find max, will identify the location as 0, containing the value 5. This value will then be swapped with the 2 in position 3.

The algorithm requires a few details. We need to keep track of what we know is sorted. Special note: the array may be sorted before the algorithm knows that it is sorted. For example, even if the the array is completely sorted, we will still go through the entire sequence of steps.

The main routine is designed to test the sorting algorithm.

```
define main {
   fillarray A
   call selection_sort
}
```

Here we give the routine **selection_sort**, which resembles the over view given above.

```
define selection_sort {
  num_rem = N
  while (num_rem > 1 ) {
      call find_max
      num_rem = num_rem -1
      call swap
  }
}
```

The variable *num_rem* keeps track of how many elements remain unsorted. Initially all elements are unsorted, so it is initialized to N, the size of the array A. After the program returns from the call to the routine **find_max**, a variable *loc* will indicate the index of the maximum value in the cells of the array between cell 0 and num_rem-1. We know that we want to reduce the value of *num_rem* before the next iteration of the loop, and we want to swap the value in $A[loc]$ with the value in $A[num\_rem - 1]$ so we reduce the variable before calling the swap routine.

Here is the subroutine **find_max** modified slightly to suit our purposes here.

```
define find_max {
  max = A[0]
  loc = 0
  i = 1
  while ( i < num_rem){
      if ( A[i] > max ) {
         loc = i
         max = A[i]
      }
      i = i + 1 # next element
  }
}
```

And to complete the program, except for declaration of variables, we show the swap subroutine, again specialized to our needs. Notice how the steps correspond to the ones in figure 6.1, although we use different names for the array indices here.

```
define swap {
    temp = A[num_rem]
    A[num_rem] = A[loc]
    A[loc] = temp
}
```

The complete maximum selection sort program can be found in the sample programs for this chapter, in the file *selectionsort.txt.*

See the exercises for some tracing problems and questions on sorting of the type you should be able to do on an exam.

6.2.3 Run time analysis of selection sort

If you try the selection algorithm for various sizes N of the array A, you will find that increasing N causes the run time and the line count **All** in the **Run Time Stats** to increase rapidly.

To check the following, you may wish to set the **Run Speed** to its maximum value of 5. For $N = 10$ you should find that the total statements executed typically ranges between 350 and 400. For $N = 20$, the range increases to 1120 through perhaps 1230. For $N = 100$ the total ranges from around 22200 to 22800, and you will also notice that it takes an appreciable amount of real time to execute, using approximately 5–6 seconds on my desk top machine with **Run Speed** set to 5. The measurement was not very accurate, as it involved watching a clock and the program at the same time.

Caution: I have had quite widely varying results in timing. Timing is not very reliable on PET.

If you try to increase the size to $N = 1000$ you can probably have a cup of coffee while waiting for a single sorting run, even with the **Run Speed** set to its maximum value.

Clearly this is a costly program to run for larger arrays. How can we get a handle on how bad it is?

Let us start at the top, in the **while** loop in the code block in **define main**. Clearly the majority of code lines executed will be in the two subroutines **find_max** and **swap**. So, we toggle the statistics on for the two lines

```
call find_max

call swap
```

which are lines 54 and 56 in the current code in *selectionsort.txt.*

If we run this after making $N = 100$ in the declaration of N, we get stats like the following (your value for **All** may vary somewhat)

Line	Number of Executions
All	22311
54	99
56	99

As an exercise you be able to see that the number of calls to each of the subroutines **find_max** and **swap** is always $N-1$ (here $99 = 100-1$). Also, every time **swap** is called, each of the 5 lines of the procedure are executed exactly once. So in total, **swap** contributes less than 500 of the total of 22311 lines executed.

Thus the majority of the time must be spent in the routine **find_max**. In this case, let us concentrate on that routine.

Recall that in our analysis of the maximum selection routine in subsection 6.1.3 we found that the key indicator of the running time of **find_max** is the number of times done in the line

```
if ( A[i] > max ) {
```

executes. This happens to be line 29 in the version of selection sort at the time of this writing. Toggling the stats on this line, and running the program with $N = 100$, we typically get **Run Tiime Stats** as follows

Line	Number of Executions
All	22755
29	4950

Again, if we try multiple runs, the line **All** will vary from run to run, but he line 29 always executes exactly 4950 times, when $N = 100$.

Where does that 4950 come from? Well, recall that when *num_rem* = 100, as it does the first time **find_max** is called, the **if** statement executes 99 times.

We saw this in subsection 6.1.3. Similarly, the next time **find_max** is called, *num_rem* = 99 and so the **if** statement is executed 98 times. Continuing in this way we see that the **if** statement is executed a total of $99+98+97+\ldots+3+2+1$ times, and if you add this up, it is indeed 4950. We also already know that the 4 lines of the **while** loop always execute every time this **if** does, so even ignoring the **if code block** the while loop always contributes $4 * 4950 = 19800$ line executions. Clearly, this is where the majority of the sorting work is done.

Can we get a better handle on this? Yes. In terms of the size of the array N that sum is just $(N-1)+(N-2)+(N-3)+\ldots+3+2+1$ and it is well known (see wikipedia e.g.) that this is equal to $\frac{N(N-1)}{2}$.

Note that $\frac{N(N-1)}{2}$ can be written as $\frac{N^2}{2} - \frac{N-1}{2}$. The term involving N^2 is the one to watch. It indicates the rough growth rate of the execution time of the algorithm. As N gets larger, the other terms become less and less important.

We say that the running time of this algorithm is $O(N^2)$ or *quadratic.*

Suppose we increase the size of the array by a factor of 10, that is, we replace N by $10N$. Then the lead term goes from $\frac{N^2}{2}$ to $\frac{(10N)^2}{2} = \frac{100N^2}{2}$. This means that increasing the size by a factor of 10 increases the running time by a factor of roughly 100.

Precisely, if we make $N = 1000$, then the number of times the **if** executes is $1000 \times 999/2 = 499500$. Since for $N = 100$ we executed the **if** 4950, this means the number of times this statement executed increased by a factor of $499500/4950 \approx 100.9$. Since the total lines executed are at last 4 times this number, we expect it to use about 2 million line executions to sort 1000 items. Here is the result of an actual run.

Line	Number of Executions
All	2030421
29	499500

We see the total number of lines executed is close to the expected. The ratio of the total lines executed of 2030421 to the 22311 in the first example above is 91, and the ratio to 22755 in the second example is 89, both of which are close to the factor of 100 we estimated by ignoring the low order terms.

This run took about 14 and 1/2 minutes. This is more than 100 times as long, which may be due to things like the larger array size, and inaccuracies in time measurement. (There seems to be a problem with a memory leak when PET is allowed to run for a long time.)

If we increased by another factor of 10, to $N = 10000$, the time should increase by another factor of about 100, or a total of roughly one day. I have neither the time nor patience to test this.

6.2.4 Insertion Sort

Like selection sort, insertion sort also maintains a part of the array that is sorted. However, unlike selection sort, the elements in this sorted part are not necessarily in their final locations until the sorting process is completed.

The idea is that we run through the items, and as we encounter each item it is inserted into the sorted part by moving over as many elements as necessary.

Figure 6.3 illustrates the result of a partial run of the algorithm. The next element that should be inserted is the 5 in location 3. To insert the 5 the elements in $A[1]$ and $A[2]$ will have to be moved one step to the right. After this insertion the first 4 elements of A would be $3, 5, 9, 11$ with $1, 8, 10$ remaining untouched.

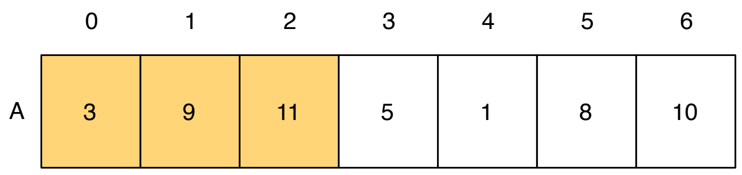


Figure 6.3: Insertion Sort with the sorted part highlighted on the left.

We could ripple each value in A to its position by repeatedly swapping it with its neighbor to the left until it is in its correct place. Here is a high level description

```
$next = 1$
while ($next < N$) {
    repeatedly swap the value in $A[next]$ with the element to its left
        until the elements $A[1] \dots A[next]$ are sorted.
    $next = next + 1$
}
```

Since the values in $A[1] \dots A[next-1]$ are sorted when we are ready to insert $A[next]$, The swapping process stops as soon as the element to the left is smaller than the value in $A[next]$.

Thus, the process above can be refined as follows

```
$next = 1$
while ($next < N$) {
    $loc = next$
    while ( ($loc > 0$ and ($A[loc] < A[loc - 1]$)) ) {
        swap $A[loc]$ with $A[loc - 1]$
        $loc = loc - 1$
    }
    next = next + 1
}
```

This swapping to the left process combines the swapping with a test to determine when the element is in its correct place. But it is frequently a good idea to break down a problem solution into its parts in order to make it clearer and to look for improvements. Recall that each swap will require three assignments to be carried out.

For the next attempt at insertion sort, we will refine the algorithm into two parts; (i) find the spot where the element should go, and (ii) move the elements bigger than the value in $A[next]$ to the right, then put the value in place.

Here is the top level subroutine **insertion_sort** from the code in the file *insertionsort.txt*.

```
define insertion_sort {
    next = 1
    while (next < N ) {
        call find_insertion_loc
        call shift_and_insert
        next = next + 1
    }
}
```

The next task in our algorithm design is to find the spot in the sorted part where the value in $A[next]$ should go. It is possible that it goes at the end of the sorted list, which means it is already in its proper place with respect to sorted part. In this case nobody moves. On the other hand, it may have to go at the very beginning, in which case all the elements in the sorted part have to move one step to the right. The following subroutine **find_insert_loc** solves this problem, but it is not very efficient. See the next subsection on analyzing this program, and the exercises at the end of the chapter for ideas on how this can be improved.

```
define find_insertion_loc {
    insert_loc = next          # the A[next] may be where it should be
    prev = next - 1
    while ( prev >= 0 ) {       # scan back to see is an earlier
                                # where A[next] should be
        if (A[prev] > A[next]) {
             insert_loc = prev
        }
        prev = prev - 1
    }
}
```

The final piece is the subroutine **shift_and_insert**. Instead of doing repeated swaps, we simply make a copy of the value $A[next]$ and then move each of the elements to the right until we reach the location where the saved value goes, then we copy it back in. Here is the code. Notice that this routine uses a total of $k + 2$ assignment operations to move the value k locations, whereas a sequence of k swaps would use $3k$ swaps, so this does save a significant number of steps.

```
define shift_and_insert {
    next_value = A[next]  # as in swap, we must save this element
    i = next
    while (i > insert_loc) {  # move elements oneto the right
        A[i] = A[i-1]
        i = i - 1
    }
    A[insert_loc] = next_value  # re-insert the saved element
                                 # in its new location
}
```

The entire program can be found in the file *insertionsort.txt*.

6.2.5 Run time analysis of insertion sort

Like selection sort, insertion sort is also quadratic, or order N squared, which we write as $O(N^2)$.

Let us get an estimate on the total number of moves required in the worst case, as they will be done in the procedure **shift_and_insert**. In the worst case, each time we insert a value $A[next]$, it must be located in position 0. So, all the items to its left must move one step to the right. Thus, when *next* is 1, 1 item moves right, and for $next = 2$ two items move and so on for a total of

$$1 + 2 + 3 + \ldots + N - 1 = \frac{N(N-1)}{2}$$

Of course, each move requires several program steps, so this number will be multiplied by a constant.

On average, using the assumption that the next value is random, we can expect that it will go in the middle of the already sorted part, which implies that the total moves will be about half of that above.

As currently implemented, a comparable number of steps will be required in the routine **find_insertion_loc** in terms the of comparisons required to find the location. In fact, as currently implemented, we continue all the way to the beginning every time, making comparisons after **insert_loc** is already established. One of the exercises asks you to fix this. But in the worst case this fix won't help.

In any case, we see that in the worst and average cases, this program requires a constant times N^2 steps to complete. In fact, as implemented, it is more costly to run on average than selection sort, requiring about 50–60% more steps on a typical run on array of size 100.

There is one bright side. As an additional exercise, think about how long this program takes when the N items in A are already sorted in increasing order. If A is already sorted, or nearly sorted, then this program will be much faster. In the "real world" things are often partially sorted, and so this might be an advantage worth considering.

However, we have much better, albeit more complex, sorting algorithms that are far more efficient. We will look briefly at one of these in chapter 7.

6.3 Binary Search of a Sorted Array

Now that we have a way to sort our arrays, we can apply binary search to the task of looking for an element.

Here again is the problem:

Instance A sorted array A of size N, and an integer x.

Query Is the integer x in the array A, and if so what is its index?

We remind again how binary search works in general.

1. We must have an *ordered search space.*
2. Secondly, we must find a question that we can ask about any particular location, and the answer must eliminate either the elements before the location or beyond the location (or both).

For this problem, searching a sorted array for an element x, first the array is sorted, so we have an ordering. Second, our query can be to compare x to $A[i]$ for any i between 0 and $N-1$. If $x > A[i]$ then if x is in A it must be in one of the cells $A[i+1] \ldots A[N-1]$. Otherwise, if $not(x == A[i])$ then if x is in A it must be in one of the cells $A[0] \ldots A[i-1]$.

Note that we must also allow for the case that x is not in A. This will happen if one of the following three occurs:

1. $x < A[0]$
   or
2. $x > A[N-1]$
   or
3. there is some i such that $A[i] < x$ and $A[i+1] > x$.

These cases might be called *boundary conditions*, because they bound the search when the value in x is not present in the array.

To implement binary search, we start by choosing upper and lower bounds that will ensure that if x is present it is between these locations. Given an array A with indices $0 \ldots N-1$ it might seem obvious that the appropriate initial bounds are 0 for the lower end and $N-1$ for the upper. Indeed, these are what we will choose for our implementation, but we note that we could approach this by choosing the lower bound to be -1 and the upper bound to be N and then working under the assumption that the range is *strictly between* these bounds. Such choices will lead to different details of implementation, for example in boundary detection. You may wish to try this as an exercise.

We use a multi-part query that guarantees that if we have not found the item in the middle, then it is either strictly to the right of the middle, or strictly to the left of the middle. By using this, and having the bounds start at 0 and $N-1$, we can cover the three boundary conditions by simply observing when the lower bound is greater than the upper bound. For example, if the lower bound is 0 and the upper bound is -1 then the upper bound is less than the lower bound, so we know the query value is not in the array. This covers the first boundary condition above, when $x < A[0]$. You should verify that testing that the lower bound is larger than the upper bound also covers the other two boundary conditions.

Here is the binary search routine that can be found in the sample programs in the file *binarray.txt*.

```
define binary_search {
    lower_bound = 0     # initally we include the entire
    upper_bound = N-1   # array as our search range
    found = false
    # we search until found or boundary conditions are false
    while ( (not found) and (lower_bound <= upper_bound) ) {
            middle = (lower_bound + upper_bound) / 2
            if (x < A[middle] ) {
                 upper_bound = middle - 1  # must be to the left
            } else {
               if (not (x == A[middle]) ) {
                    lower_bound = middle + 1 # must be to the right
               } else {
                   found = true
               }
            }
     }
     if (found) {
        writeline x " is found in location " middle
     } else {
        writeline x " is NOT in the array."
     }
}
```

Note that we only report on whether or not we found the item after the while loop is complete. We simply have to check the boolean value $found$ to know whether or not it was found in the array. If it was found, then the index *middle* indicates its location.

6.3.1 Binary Search analysis

As in subsection 5.3.2 we note that if we make k queries where $2^k \geq N$ then we will know whether or not our number is in the array. To test this, use the code in the file *binarray.txt* and modify it to set $N = 100$ as the initial array size. Note that $2^7 = 128 > 100$ so we should generally find the number if it is in A in 7 iterations of the while loop, and otherwise we will know it is not in the array.

Now toggle on the statistics for the line

```
        middle = (lower_bound + upper_bound) / 2
```

Make a number of queries for numbers both in and not in the array, and note that we never increase the stats on this line by more than 7 for each query.

6.4 Exercises on Searching and Sorting

In the following, when a question refers to an array of size N, then you should make your program so that it works for any size N. You should try it for $N = 10$, $N = 25$ and a few other sizes to be certain it works correctly. All programs to be written in PET.

1. **Vector addition** Write a program that takes two arrays of the same size N, and adds the values pairwise into the corresponding cells of a third array. Try your program by randomly filling the two input arrays using fillarray.

2. **Squaring numbers** Write a program that starts with an array of size N. Fill the array with the squares of the index of the corresponding cell, for example $A[3]$ should have value 9.

3. **Fibonacci** Write a program that fills an array A of size N as follows. $A[0] = 1$, $A[1] = 1$ and for $i > 1$, $A[i]$ is the sum of $A[i-1]$ and $A[i-2]$.

4. **Running Totals** Write a program that fills an array of size N so that the $A[0] = 1$ and for $i > 0$, $A[i]$ has the total of cells $A[0]$ through $A[i-1]$. Are you surprised by the result?

5. **Smallest Remainder** Randomly fill an array of size $N = 25$ and then find the element that has the smallest remainder when divided by 131. Recall we developed a remainder function in chapter 5.

6. **Make Find Efficient** Modify the program in the file *seqfind.txt* so that as soon as the hidden value is located the while loop terminates, instead of running on through the rest of the array.

7. **Count Finds** Modify the program in the file *seqfind.txt* so that it counts how many times a number occurs in an array. If the number is not in the array, then your program should report 0 times. To test this, after filling the array, replace each entry in the array with its remainder when divided by 10. If you have an array of size $N = 25$, some numbers will be repeated. Now modify the program so that it asks the user for the number to be searched for. On running the program, since you can see the array, you ask for those numbers and verify your program counts correctly.

8. **Min** Write a program that fills an array of size N, and then find the minimum value in the array and report its location. Note this is very similar to the find the maximum problem.

9. **Min Variation** Write a program that randomly fills two arrays A and B each of size N. Your program should report the smallest value in B that is larger than the smallest value in A. For example, if A has the values $4, 11, 7, 13, 5$ and B has values $8, 2, 5, 3, 11$ then your program should output 5.

10. **Counting Line Executions** Starting with the code from *findmax.txt*, using $N = 25$, add the following line immediately after the line `fillarray A`

    ```
    A[0] = 10000
    ```

    Note that $10000 > 250$, the maximum value that will be assigned by **fillarray**. Without running the program, predict how many times will the line `loc = i` be executed? Verify your answer by running the program.

    Note: when you insert or delete lines of code in PET, old toggles will no longer be on the correct lines. Be sure to change them. It is best to toggle them off before editing.

11. **More Counting** Starting with the code from *findmax.txt*, using $N = 25$, replace the line `fillarray A` with the following code

    ```
    i = 0
    while ( i < N) {
       A[i] = i
       i = i + 1
    }
    ```

    Predict how many times the statement `loc = i` will be executed, and verify your prediction.

12. **Tracing Selection Sort** In the routine **selection_sort** in the code in the file *selectionsort.txt* we will save a *phase* completes each time the program completes execution of the line "**call swap**". The first row of the following table represents the array A of size 7 just after the array has been randomly filled. Complete the table by filling in the rows showing the state of the array after the completion of each phase until the program terminates.

13	2	7	5	14	1	9

Note: to verify your answer, you can modify the selection sort program so that it uses an array of size 7, and asks for the array to be input by the user instead of being generated at random. Then toggle on a break point immediately after the line "**call swap**" and check the results in memory against your answers here.

13. **Minimum Selection Sort** Modify the selection sort routine to work by repeatedly selecting the minimum value in the unsorted part and inserting it at the beginning of the unsorted portion. Thus, in this version the sorted part is at the beginning, and the unsorted is at the end. Note this does **not** mean the array is to be sorted in descending order. The final array should be in ascending order.

14. **Tracing Insertion Sort** In the routine **insertion_sort** in the code in the file *insertionsort.txt* we will save a *phase* completes each time the program completes execution of the line "**call shift_and_insert**". The first row of the following table represents the array A of size 7 just after the array has been randomly filled. Complete the table by filling in the rows showing the state of the array after the completion of each phase until the program terminates.

13	2	7	5	14	1	9

Note: to verify your answer, you can modify the insertion sort program so that it uses an array of size 7, and asks for the array to be input by the user instead of being generated at random. Then toggle on a break point immediately after the line "**call call shift_and_insert**" and check the results in memory against your answers here.

15. **Improving Insertion Sort (I)** Modify the subroutine **find_insertion_loc** in the program in the file *insertionsort.txt* so that once we know for certain what the insertion point is, the loop is terminated, instead of running all the way to minus 1 each time. Run some tests to see if you can notice any improvement.

16. **Improving Insertion Sort (II)** Modify the subroutine **find_insertion_loc** in the program in the file *insertionsort.txt* to use binary search to find the location of the next insertion point. Run some tests to see if you can notice any improvement.

17. **Using Sorting** When **fillarray** is used to fill an array, since the numbers are generated (pseudo) randomly for each entry, it is possible that some number appears more than once in the array.

    (a) Write a subroutine in PET code to determine whether or not any number appears more than once in an array A. Hint: start with one of the sorting programs, and make it into a subroutine. Make your program work on the sorted array.

    (b) Once you have the subroutine in the first part working, write a program to repeatedly fill an array 100 times and determine whether or not it contains duplicates. For an array A of size $N = 10$ what fraction of the filled arrays contain no duplicates? Note your answers will likely vary over repeated trials. In science, once we introduce chance elements answers to questions like this get messy.

Chapter 7

More Algorithm Design

In this chapter we extend the algorithms theme, looking at more complex algorithms with the goal of illustrating how design improvements can affect the running times dramatically. Different professors may wish to do different portions and give different emphasis in this part of the course, or perhaps present different material altogether. Students are advised to check what material is being covered for the last few lectures in their section of the course, as it may not be the same as other sections.

7.1 MergeSort

There are many sorting algorithms that we have not considered. Much effort has been spent over the years in making algorithms that are more efficient in terms of time and memory used. In an introductory course such as this we mainly choose algorithms that are reasonably easy to present within the context of the tools at hand, and which show some underlying principles of algorithm design.

We judge mergesort to be one of the easier of the $O(N \log N)$ (see subsection 7.1.4 for what that means) algorithms to understand and explain at this level, although it is neither the fastest nor best in terms of memory usage in practice. In addition, the bottom up natural mergesort that we use is relatively easy to implement using PET, whereas a program that used recursion would be more difficult given that PET does not support parameter passing as part of the language. Finally it is a good case to study in terms of how seemingly innocuous design decisions can in fact have a large impact on the efficacy of the program.

The sorting algorithms previously considered, selection sort 6.2.2 and insertion sort 6.2.4, maintained a sorted set that was added to one element at a time from the unsorted portion of the array. In this section we consider an algorithm that works by creating multiple sorted subparts of the array, and then merging them in pairs. This process repeats until it is all sorted.

Unlike the previous sorting algorithms, this algorithm is not “in place” meaning that we need extra arrays to store sorted pieces before merging them back into the original array. But there is a pay back for this extra memory; this algorithm is much faster for larger arrays, and the bigger the arrays get, the more it outperforms these other two algorithms.

The general idea of our implementation of merge sorting can be summarized like this:

```
while not sorted {
    find two adjacent sorted subparts in $A$ (called *runs*)
    copy these runs to two separate arrays
    merge these back into the array $A$ replacing
        the two runs with one combined run
}
```

The rest of this section will develop these ideas in detail. We assume throughout that the array we want to sort is called A and that we have two additional arrays B and C.

7.1.1 Finding and copying the runs

The first step is to identify a run. *A* **run** *is defined to be a maximal consecutive sequence of increasing values in the array.*

The first run will start at location 0, that is $A[0]$. It will continue until we reach an index i such that $A[i] > A[i+1]$ or until $i+1$ is equal to N, the number of elements in A. If $i < N-1$ then the next run will begin at $i+1$.

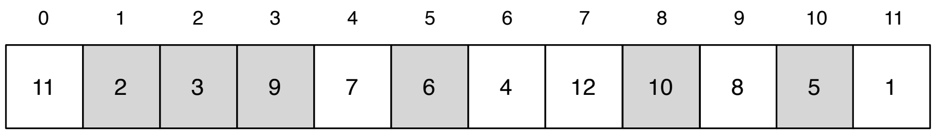


Figure 7.1: An A with runs indicated.

Figure 7.1 illustrates an array with 12 elements and 9 runs. The first run consists of the element $A[0] = 11$. It is maximal because the next element in $A[1]$ is 2 which is less than 11. The second run is in $A[1], A[2], A[3]$ with values $2, 3, 9$. The remaining runs are indicated by sequences of boxes of the same shade.

How do we identify and copy a run? We will know which element is the first since it will either be $A[0]$ or the next element after the previous run, so all we have to do is copy items until either we reach the end, or until $A[i] > A[i+1]$ which ends the run.

Here is the subroutine for copying a run starting at index **index_a** in A into array B. The Boolean variable **flag** is used to stop the copying when the run ends. Note that we initialize **flag** so that if **index_a** is already at the end of the array nothing is copied. Otherwise, at least one value will be in the run, and so if we are not beyond the end of the array, we copy a run of at least one element. The variable **number_in_B** indicates how many items we copy into B, that is, it is the length of the run. It also gets used as the index of where to put the next element of the run in B. When the subroutine completes, **index_a** will be at the beginning of the next run, or equal to N if the run goes all the way to the end of A.

```
define copyrunB {
   number_in_B = 0
   flag = ( index_a < N)
   while ( flag ) {
     B[number_in_B] = A[index_a]
     index_a = index_a + 1
     if ( index_a >= N ) {
        flag = false   # end of the array
     } else {
        if (B[number_in_B] > A[index_a] ) {
           flag = false  # next value is smaller so end
        }
     }
     number_in_B = number_in_B + 1
  }
}
```

This routine, and a very similar one called **copyrunC** for copying a run into array C, can be found in the file *mergesort.txt*.

7.1.2 Merging two sorted arrays

Figure 7.2 illustrates how the first two runs should be first copied out into arrays B and C and then merged in order back into array A. We saw previously how the copy would be done.

Here is a high level view of how the merge of two arrays should be done:

```
start an index for each array indicating
the next element in that array
while there are elements in both arrays {
     copy the smaller indicated value
     and advance that indicator
}
copy the elements from the array that
was not all copied above
```

Figure 7.3 illustrates a merge process partially completed. Three elements have already been copied back into A, two from B and one from C. The next step will compare the two values 6 in B and 5 in C. Since 5 is smaller, it will be copied into the indicated position in A, and **index_a** and **index_c** will each be incremented by one.

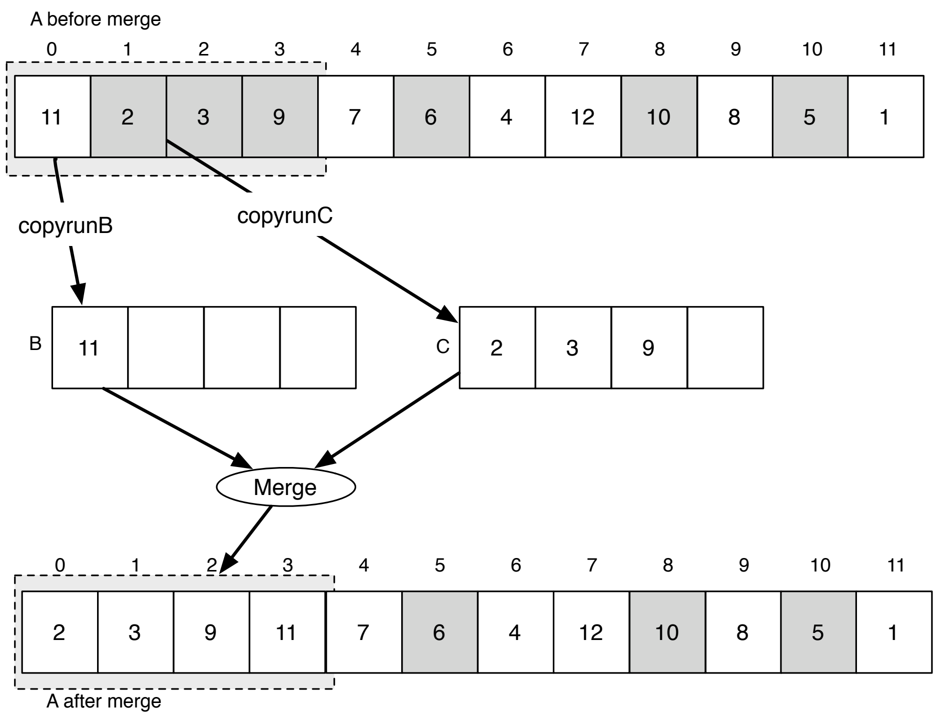


Figure 7.2: Two runs are copied, and then merged back into A.

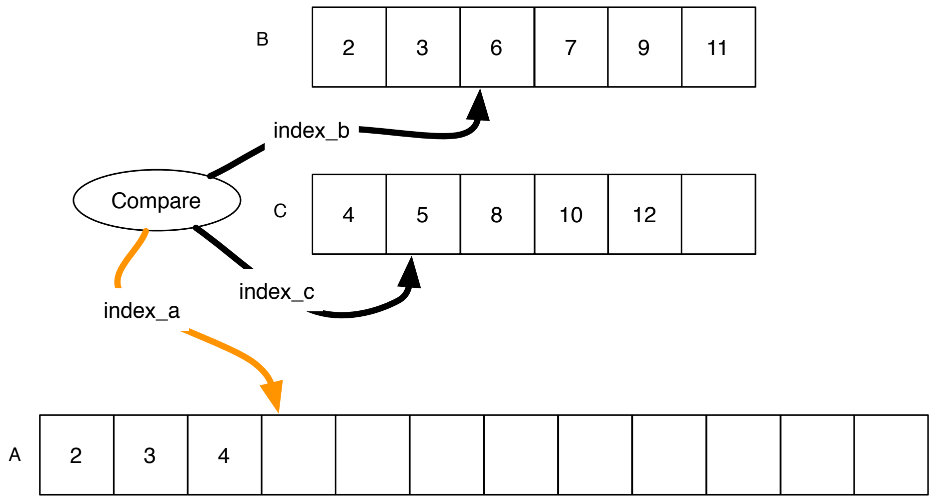


Figure 7.3: One step of a merge into A.

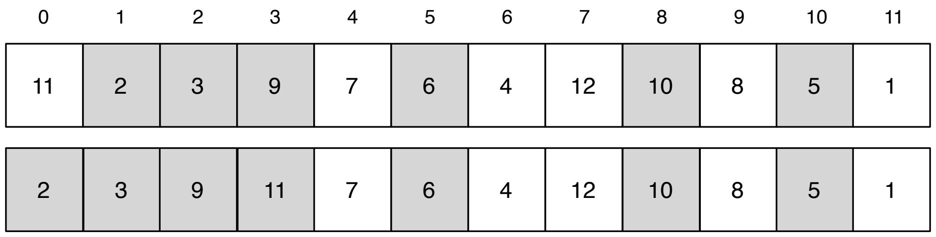
If we continue this process, eventually the value 11 in B and 12 in C will be compared, and the 11 will be copied into A. At that point no further elements remain from the run in B. Since the values in C are sorted, they can be copied in order from C into A. In this case there will only be the one value left in C, but in general there could be any number left.

Below is the implementation of this algorithm in PET which can be found in the file *mergesort.txt*.

```
# Merge the contents of arrays B and C
# back into A starting at the current index_a
define merge {
   index_b = 0
   index_c = 0

   while (( index_b < number_in_B) and (index_c < number_in_C) ) {
     if (B[index_b] <= C[index_c] ) {
        A[index_a] = B[index_b]
        index_b = index_b + 1
     } else {
        A[index_a] = C[index_c]
       index_c = index_c +1
    }
    index_a = index_a + 1
  }

  # clean up the list that was not emptied above
  # Note: Only one of the following while loops
  # will execute on any given merge

  while ( index_b < number_in_B) {
     A[index_a] = B[index_b]
     index_a = index_a + 1
     index_b = index_b + 1
  }

  while ( index_c < number_in_C ) {
     A[index_a] = C[index_c]
     index_a = index_a + 1
     index_c = index_c + 1
  }
}
```

7.1.3 Putting it together: Natural MergeSort

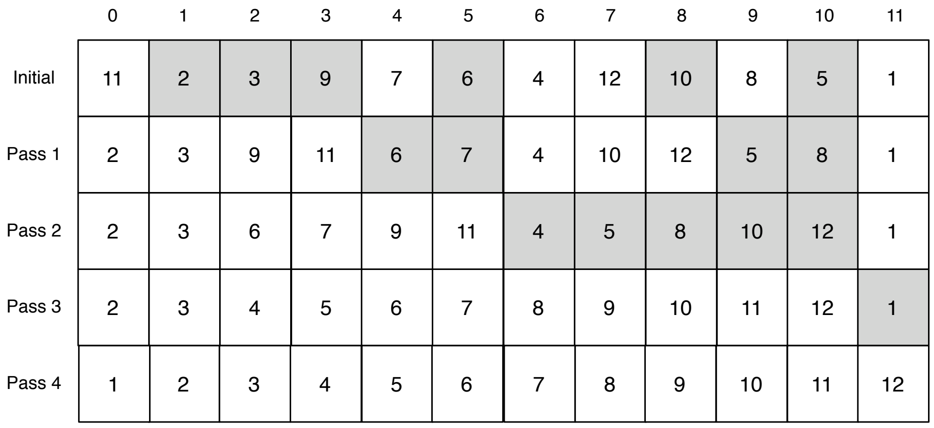
We now know how to find runs, copy them out, and merge them back into A. But a very important design decision remains, and that is in what order do we do the merges? It seems fairly clear that we can start by merging the first two runs of the array.



0	1	2	3	4	5	6	7	8	9	10	11
11	2	3	9	7	6	4	12	10	8	5	1
2	3	9	11	7	6	4	12	10	8	5	1

Figure 7.4: A after the merger of the first two runs.

Figure 7.4 illustrates the original A and the result after the first two runs are merged. Now comes the crucial design decision. We could at this point again merge the first two runs, namely the run of 4 cells $A[0] \dots A[3]$ and the run of one element in $A[4]$. After that we can again merge the new first run with the run in $A[5]$ and so on until all the runs are merged. You are asked to try this technique in the exercise at the end of this chapter labelled **Importance of Order**. It turns out this is **not a good idea**, as you should discover if you try the exercise.



	0	1	2	3	4	5	6	7	8	9	10	11
Initial	11	2	3	9	7	6	4	12	10	8	5	1
Pass 1	2	3	9	11	6	7	4	10	12	5	8	1
Pass 2	2	3	6	7	9	11	4	5	8	10	12	1
Pass 3	2	3	4	5	6	7	8	9	10	11	12	1
Pass 4	1	2	3	4	5	6	7	8	9	10	11	12

Figure 7.5: A trace of mergesort showing the result after each pass.

Instead, we will do merges in *passes*. On each pass we merge the current set runs in pairs. This is illustrated in figure 7.5, where the state of the array is shown after each pass. For example, after *pass 1* the first two runs are merged into the run $2, 3, 9, 11$ the next two runs are merged into $6, 7$, the next two create $4, 10, 12$ and the next two create $5, 8$. Notice that initially there were an odd number of runs, and so the run consisting of 1 remains unchanged since there was no run to pair it with.

On the next pass two merges are done, creating runs $2, 3, 6, 7, 9, 11$ and $4, 5, 8, 10, 12$ and again leaving 1 isolated because there were an odd number of runs. Pass 3 merges all but the last element, and pass 4 completes the sorting.

All that is left are details of implementation. First, when do we know we are done? Well, if there is only one run in A that means the array is sorted, so we check to see if on the last pass all elements are copied into B. If we were a bit more clever we could save one pass by noting when the previous pass created only one run, but that makes the implementation more complex, and our goal here is simplicity of presentation over ultimate efficiency. You may want to think about this as an advanced exercise. In relation to the diagram in figure 7.5 our poor implementation makes one more pass to notice that the array is sorted.

Next we need to keep merging pairs until the pass is complete. There are a few details of remembering where the next pair starts after we complete the two copies, which we document in the code below.

```
define mergesort {
   number_in_B = 0
   # as suggested in the notes, the test used in the
   # following while is simple to understand, but the program
   # will waste a pass after finishing sorting.
   while ( number_in_B < N ) {
     start = 0
     # start remembers where the next pair of runs begins
     # when it reaches N, the pass is complete
     while ( start < N ) {
        index_a = start
        call copyrunB
        call copyrunC
        # a little trick, we have to reset index_a
        # before calling merge, but we do not want to
        # forget where the next pair starts,
        nextstart = index_a
        index_a = start
        call merge
        start = nextstart
     }
   }
}
```

7.1.4 How fast is mergesort?

We say that mergesort sort (as implemented here) is $O(N \log N)$. What does that mean and why is it true?

Recall that if k is the smallest integer such that $2^k \geq N$ then we say k is (approximately) $\log_2 N$. And as before we use the *order* (or big-O) notation $O(N \log N)$ to mean that the running time is roughly a constant times $N \log N$, which is to say a constant times Nk where $2^k \approx N$. ($\approx$ means approximately equal to.)

If you think about it, this means that the running time is growing just a little faster than proportional to N. For example, if $N = 1000$ then $k \approx 10$, and so $N \log N \approx 10,000$. So the running time is some constant times 10,000. (The constant may be smaller than 1.) If we let N grow to a million then k is approximately 20, so the running time increases by approximately a factor of $\frac{1000000 \times 20}{1000 \times 10} = 2000$. Since going from an array of size 1000 to an array of size 1000000 is a factor of 1000, you can see that here the growth is twice linear. But it is significantly less than $O(N^2)$.

Here is a table comparing N, N^2 and $N \log N$ for a few values.

N	$\log_2 N$	$N \log_2 N$	N^2
128	7	896	16384
1024	10	10,240	1,048,576
4096	12	49,152	16,777,216

In summary, $O(N \log N)$ is significantly better than $O(N^2)$ as the arrays get larger. For example, for $N = 4096$ we see that $N \log_2 N$ is more than 340 times smaller than N^2.

But why does it have this behavior? Look again at figure 7.5. Consider the number of runs at the end of each pass. Initially there are 9 runs, then 5, then 3, then 2 and finally 1. If we ignore the odd one left over at the end, this becomes 8, 4, 2, 1 (and finally 0). But these are our old friends, the powers of 2. Why is this? Well, on each pass we merge each successive pair of runs into one. So, on each pass we cut the number of runs in half (except when there is an odd one left over). Now, clearly the initial number of runs cannot be more than N, since there are only N elements. (You should see the exercise labeled **Worst Case Runs** to learn when the number of runs is exactly N.) Now using the same argument we have used before, if $2^k \geq N$, we cannot cut N in half more than k times until there is only one run, and then of course the array is sorted. So we can make at most k passes before the array is sorted.

Now we only need to consider the work done on each pass. But this is easy. On a pass every element of A is first copied to either B or C, then some comparisons are done (at most one per element) and finally the element is copied back to A. So the total number of steps is at most some constant times N times k, where k is $\log_2 N$.

For arrays of size $N = 100$ this mergesort takes about 12700 steps. Compare this to the 22600 used by selection sort, or the more than 30000 used by our original insertion sort, and it looks quite good. When we increase the array size to $N = 1000$ our mergesort typically takes approximately 175700 steps in total which is less than 1/10 the number that selection sort typically takes.

A final note: Importance of Order

In the exercise labeled **Importance of Order** and the discussion in section 7.1.3 we observe that as an alternative algorithm we could repeatedly merge the first list with the one that follows, but claimed that this was a bad idea. Briefly here is why it is a bad idea

Suppose that the initial order is such that every element is in a run of size one (see the exercise labeled **Worst Case Runs**). For every merge, we first copy the two lists, one to B and the other to C, and then we copy them back. But if every run is of size one, then under this ordering of merges, every merge increases the first run length by 1. For the first merge, we copy 1 into B and back, then the next time 2, then the next time 3 and so on. The total number of elements copied to B and back is thus $1+2+3+\ldots+N$ which, as we have seen several times, is order N squared, i.e. $O(N^2)$. So as N gets large this is much less efficient than merge sort as we implemented it. Of course, initially the runs are not all of length one, but the average length of a run initially is typically a small constant or less, so this will only reduce the run time by a small constant, which means this bad order version is still $O(N^2)$ for random arrays.

Notice how this analysis complements and extends our understanding of the sorting algorithm. Such depth is a key part of algorithms and computing science.

7.1.5 Merge Sort Questions

1. [**Worst Case Runs**] Construct an array with 10 elements in which there are 10 runs. In general, what property has to be true for an N element array to have N runs?

2. [**Importance of Order**] Modify the **mergesort** subroutine in the file *mergesort.txt* so that instead of merging the runs in pairs through a sequence of complete passes, it instead repeatedly merges the run that starts in $A[0]$ with the one immediately following it. See the discussion of figure 7.4 in subsection 7.1.3 to clarify what is required. Note the program actually gets simpler.

   For each of $N = 100$ and $N = 200$, compare the number of statements used by the two versions of mergesort, doing several tests at each size. See the analysis at the end of section 7.1.4 for an explanation.

3. [**Merge Extension**] Write an algorithm that merges 3 sorted lists. Note this does not ask for merge sort, just a merge. But if you are adventuresome, feel free to write a merge sort that works by merging sets of up to three runs.

4. [**Understanding Merge(trace)**] What happens if we run the merge algorithm on two lists that are not sorted? Suppose $B = [2, 4, 7, 13, 5, 8]$ and $C = [3, 9, 6, 21, 1]$. Show the result of running the merge algorithm on these two arrays. Note you are only running one merge, not merge sort. Assume **index_a = 0** when you start.

5. [**Trace Merge Sort**] Identify the runs in the following array, then trace the execution of Merge Sort, showing the state to the array after each complete pass as done in class. Identify the runs as each pass ends. Note, you may not need the entire diagram. (You can ignore the wasted last pass in the on line implementation - see the following exercise).

13	2	7	5	15	14	9	1

6. [**Cosmetic Improvements**] As it is currently implemented, there are a number of improvements one might make to the program in the file *mergesort.txt*. Notice that the program terminates after it notices that the last iteration copied the entire array into the array B then copied it back. This is wasteful because it copies the entire array into B and then back into A after it is already sorted. Modify the program so that if the last pass sorts the array, it stops without this wasteful pass. What, if anything, can you do about the special case when A is sorted before the program starts?

   We call this a *cosmetic improvement* because although it reduces the total cost of sorting, it does not change the $O(N \log N)$ worst case, but only reduces the cost by $O(N)$ work. Still, in practice this can be worthwhile.

7. [**More Cosmetics**] In the file *mergesort.txt* the program uses two arrays B and C, and both of these must be the size of A since we cannot predict how long a run may be. However, it is possible to use just one extra array B. Since two sequential runs can never exceed the length of A, this array need only be the size of A. The idea is to copy both runs into B. We just need to keep track of where the run copies are located in B, then merge them back to A. Modify the program so that only one extra array is required. You may need to keep track of a few more indices.

Note: trying to make merge sort use as little extra memory as possible has been the subject of research for some time.

7.2 Algorithm Design Example

While you are using a computer, you will open and close various programs, such as email, web browsers, editors, program tools such as PET and BeeperBot, and many more. Each of these requires memory while running with different amounts required at different times. The *operating system* you are using, such as Unix, or Windows, or MAC OS X has to keep track of what memory is in use, and what is available. In fact it is more complicated than this discussion might imply, but as a simplification, you can think of this as a list of entries saying what memory is being used by each program. Suppose it is stored in an array (the reality is more complicated). We can think of each entry as being the amount of memory being used. When an entry goes away, the entry becomes 0.

We would like from time to time to go through the array and "remove" the entries that are zero, so that keeping track of the remaining entries is easier.

A similar problem occurs if we are taking data samples from a large array of sensors. Some of them may not report at every time period, and so these could be given a special value, say 0 if that does not conflict with what an actual data reading might be. Again, to save memory, it could be useful to go through a large array and "remove" all the entries that are not desired.

But what does removing an entry of 0 mean? Well, it means we want to copy all the entries that are not 0 to these locations, so that the k non-zero entries are in cells $A[0] \dots A[k-1]$.

Here is an example where the first row shows an array with zero entries, and the second shows the same data moved to the beginning, with the blanks at the end indicating we do not care what is in those entries.

0	8	7	0	11	2	0	0	5	9	0	11
8	7	11	2	5	9	11					

This problem will be presented as a series of exercises that you should be able to implement. To get you started, there is a program in the online set of programs in a file called *dataclean.txt* which creates a sparse array, that is one in which many of the entries are zero. For each of the following, you should start with an array of size 10, and use that to verify your program works correctly by keeping all the non-zero values, and eliminating all the zeroes that occur in the first k elements of the array. Then run it on larger arrays of size 100 to 200 to see how the run time changes according to the algorithm.

1. The first idea is the easiest, but actually uses more memory, because we use another array. Nevertheless, the data will be compact in B. Create another array B of the same size as A, and simply copy the k non-zero items from A to the first k entries of B.

   Since we only go through A once, this program runs in time $O(N)$.

2. The next version does not require an extra array, but requires $O(N^2)$ time in the worst case. It is fairly simple, with one minor twist. Use an index, say i, to traverse the array. If $A[i]$ is not zero, then simply increase i. But when $A[i]$ is zero, then use another index say j to run through the rest of the array and move every element one position towards i. The twist: do not change i until you check whether $A[i]$ is still zero!

   Why is this $O(N^2)$? Consider how many moves there are in total if every entry is 0.

3. In this version, we again scan the array with index i. When we he find an i such that $A[i]$ is zero, we scan ahead until we find a non-zero entry (or reach the end of the array, in which case we are done.) If the non-zero value is at position j we copy it to $A[i[$ and then set $A[j]$ to zero to prevent copying it again.

   As presented this version is also $O(N^2)$. Consider what happens when the first $N/2$ entries of A are zero, and the remaining $N/2$ entries are valid values. The issue is that we keep running the second index over the locations containing zero over and over again. The next exercise asks you to fix this.

4. You should now change the previous version so that when we hit a zero entry, and we copy an item from $A[j]$ to $A[i]$, we remember j. Now when i hits another zero, j scans from where it left off last to find the next non-zero value.

   Since now each of i and j go over the array at most once, it does work proportional to N and so this algorithm is $O(N)$.