An Introduction to Programming with ECMA-55 Minimal BASIC

John Gatewood Ham

August 30, 2021

Acknowledgments

I would like to thank Narongsak Chusinchinnapat for his in-depth technical review of a draft of this book. I would also like to thank Kanjana Eiamsaard for her feedback on first drafts of two chapters of this document. I would also like to thank Arthit Archdet for submitting a homework program that proved I had not correctly implemented array subscript rounding. I also want to thank the students from Laos who used drafts of the book in a classroom environment:

- Thitmany Sisavath
- Vilaikone Phanthasomchit
- Khamkhen Vilayphone
- Syvixay Phachansily
- Soulisack Bounmiphone
- Saythong Phamoungkhoun
- Khankham Lorkhamxay

Finally, I would like to thank Doug Kearns for his bug reports and patches.

Contents

Page

List of Figures **iv**

List of Tables **ix**

Preface **xi**

1 Introduction **1**

2 HELLO, WORLD! **5**

3 Temperature Conversion **9**
3.1 Convert Celsius to Fahrenheit 9
3.2 Convert Fahrenheit to Celsius 11

4 Scalar Variables and Constants **15**

5 Generating Sequences **21**
5.1 Loops 24
5.2 FOR Loops 31
5.3 Summary 33

6 More Series **37**
6.1 Factorials 37
6.2 Fibonacci Numbers 43
6.3 Taylor Series 47

7 Random Numbers **51**

8 Multi-way Branching **57**
8.1 Implementation of multi-way branch using `IF` 58
8.2 Implementation of multi-way branch using `ON...GOTO` 60

9 Multicolumn Output **63**
9.1 First Draft 63
9.2 Second Draft 66
9.3 Third Draft 69
9.4 Final Program 72

10 Arrays **79**
10.1 Average Value 80
10.2 Maximum Value 83
10.3 More About Subscripts 87

11 Including Data Within A Program **91**
11.1 A simple example using `READ` and `DATA` 91
11.2 Three ways to read lists of data 94

12 Sequential Search **99**
12.1 The Algorithm 99
12.2 Implementation in ECMA-55 Minimal BASIC 104

13 Subroutines **107**

14 Bubble Sort **113**
14.1 The Algorithm 113
14.2 Implementation in ECMA-55 Minimal BASIC 118

15 Binary Search **121**
15.1 Binary Search Example 1 122
15.2 Binary Search Example 2 123
15.3 Binary Search Example 3 124
15.4 Binary Search in Detail 126
15.5 Performance of Binary Search 127
15.6 Implementation in ECMA-55 Minimal BASIC 130

16 Two-Dimensional Arrays **133**

17 User-defined Functions **145**

18 Singly Linked Lists **149**
18.1 The Dynamic Memory Concept 150
18.2 Storing a linked list in a matrix 153
18.3 Traversing the linked list 155
18.4 Initializing the heap 157
18.5 Accessing an individual node 159
18.6 Updating field values of a node 160
18.7 Appending a node to a linked list 162
18.8 Removing a node from a linked list 167
18.9 Implementation in ECMA-55 Minimal BASIC 172

19 Summary **179**

A The bas55 Interpreter **181**

B The ecma55 Compiler **185**

C BASIC Statements **187**

D BASIC Numeric Functions **197**

E Flowcharts **205**

F GNU Free Documentation License **219**
1. APPLICABILITY AND DEFINITIONS 219
2. VERBATIM COPYING 221
3. COPYING IN QUANTITY 222
4. MODIFICATIONS 222
5. COMBINING DOCUMENTS 224
6. COLLECTIONS OF DOCUMENTS 225
7. AGGREGATION WITH INDEPENDENT WORKS 225
8. TRANSLATION 226
9. TERMINATION 226
10. FUTURE REVISIONS OF THIS LICENSE 226
11. RELICENSING 227
ADDENDUM: How to use this License for your documents 227

Index **229**

List of Figures

Page

Figure 1.1: Sample Flowchart 2
Figure 2.1: Hello World Flowchart 5
Figure 2.2: Hello World Program 5
Figure 2.3: Hello World Program Output 6
Figure 3.1: Flowchart for Converting °C to °F 10
Figure 3.2: Program Celsius to Fahrenheit 10
Figure 3.3: Program Celsius to Fahrenheit Output for 0°C 10
Figure 3.4: Program Celsius to Fahrenheit Output for 43°C 10
Figure 3.5: Flowchart Symbols 11
Figure 3.6: Flowchart for Converting °F to °C 12
Figure 3.7: Program Fahrenheit to Celsius 12
Figure 3.8: Program Fahrenheit to Celsius Output for 0°F 12
Figure 3.9: Program Fahrenheit to Celsius Output for 43°F 12
Figure 4.1: Scalar Variables are Addresses 15
Figure 4.2: Program Fahrenheit to Celsius with Comments 17
Figure 5.1: Flowchart 1 22
Figure 5.2: Program 1 22
Figure 5.3: Program 1 output 22
Figure 5.4: Flowchart 2 23
Figure 5.5: Flowchart 3 24
Figure 5.6: Flowchart 4 25
Figure 5.7: Flowchart 5 27
Figure 5.8: Program Source Version 1 27
Figure 5.9: Program Output 28
Figure 5.10: Flowchart 6 29
Figure 5.11: Program Source Version 2 29
Figure 5.12: Flowchart 7 30
Figure 5.13: Program Source Version 3 31
Figure 5.14: Program Source Version 4 32
Figure 5.15: Program Source Version 5 33
Figure 5.16: Output of Version 5 Program 33
Figure 6.1: Factorial Series Flowchart 38
Figure 6.2: Factorial Program Source 39
Figure 6.3: Factorial Program Output 39
Figure 6.4: Nicer Factorial Series Flowchart 42

Figure 6.5: Best Possible Factorial Output 43
Figure 6.6: Ugly Factorial Output 43
Figure 6.7: Nicer Factorial Program Source 43
Figure 6.8: Fibonacci Sequence Flowchart 45
Figure 6.9: Fibonacci Sequence Program Source 45
Figure 6.10: Fibonacci Sequence Program Output 46
Figure 6.11: Taylor Series Cosine Source 48
Figure 6.12: Taylor Series Cosine Flowchart 49
Figure 7.1: High-low flowchart 51
Figure 7.2: High-Low Program 52
Figure 7.3: Improved High-low flowchart 53
Figure 7.4: Improved High-Low Program 54
Figure 7.5: Improved High-Low Program Output 54
Figure 8.1: Five-way branch flowchart 57
Figure 8.2: Five one-way branches flowchart 58
Figure 8.3: Five one-way **IF** statements 59
Figure 8.4: Five one-way **IF** statements (alternate solution) 59
Figure 8.5: **ON** . . . **GOTO** statement 60
Figure 8.6: One five-way **ON**. . . **GOTO** statement 60
Figure 9.1: Five column program flowchart 64
Figure 9.2: Program for five column output 65
Figure 9.3: Five column output when not a multiple of 5 65
Figure 9.4: Five column output for a multiple of 5 65
Figure 9.5: Improved five column program flowchart 67
Figure 9.6: Improved program for five column output 68
Figure 9.7: Even better five column program flowchart 70
Figure 9.8: Even better program for five column output 71
Figure 9.9: Five column failure 71
Figure 9.10: Full-featured multicolumn program flowchart (1 of 2) 74
Figure 9.11: Full-featured multicolumn program flowchart (2 of 2) 75
Figure 9.12: Full-featured multicolumn program 76
Figure 9.13: Full-featured Multicolumn Program Output 77
Figure 10.1: Zero-based Array A with 9 elements 79
Figure 10.2: Compute Average Value Flowchart 81
Figure 10.3: Compute Average Value 82
Figure 10.4: Compute Average Value Program Output 82
Figure 10.5: One-based Array A with 9 elements 83
Figure 10.6: Find Maximum Value Flowchart 84
Figure 10.7: Find Maximum Value 86
Figure 10.8: Find Maximum Value Program Output 86
Figure 10.9: Use of non-integral subscripts 87
Figure 10.10: Use of non-integral subscripts2 88

Figure 10.11: Using logical non-integral subscripts 89
Figure 11.1: Load Array With Data Flowchart 92
Figure 11.2: **READ** and **DATA** example 93
Figure 11.3: **READ** and **DATA** Program Output 93
Figure 11.4: Example of reading a list of data three different ways 95
Figure 11.5: Output of program in figure 11.4 96
Figure 12.1: Sequential search for X=5 in array A 100
Figure 12.2: Sequential search for X=-1 in array A 101
Figure 12.3: Sequential search for X=4 in array A 101
Figure 12.4: Sequential Search Flowchart for N elements 103
Figure 12.5: Sequential Search Program 105
Figure 13.1: Subroutine Example Flowchart (1 of 2) 108
Figure 13.2: Subroutine Example Flowchart (2 of 2) 109
Figure 13.3: Subroutine Example Program 110
Figure 14.1: Bubble sort flowchart for N elements 117
Figure 14.2: Bubble sort program 119
Figure 15.1: Flowchart for iterative binary search of array of N elements . . 127
Figure 15.2: $O(n)$ Graphs 129
Figure 15.3: Binary search program 131
Figure 16.1: Layout of a zero-based 6x4 matrix 133
Figure 16.2: Layout of a one-based 5x3 matrix 134
Figure 16.3: main, load_data, and get_item_number 134
Figure 16.4: get_price subroutine 135
Figure 16.5: get_quantity subroutine 136
Figure 16.6: run_menu subroutine 137
Figure 16.7: print_report subroutine 138
Figure 16.8: print_product_name subroutine 139
Figure 16.9: Logical Array of Records program (1 of 2) 141
Figure 16.10: Logical Array of Records program (2 of 2) 142
Figure 17.1: Example Using User-defined Functions 145
Figure 17.2: User-defined Functions Program Output 146
Figure 18.1: Singly Linked List 149
Figure 18.2: Pointer variables contain address that can be changed.
Scalar variables are aliases for constant addresses. 151
Figure 18.3: Free List 151
Figure 18.4: Singly linked list in a matrix 153
Figure 18.5: `dump_raw_storage` procedure 154
Figure 18.6: `print_nodes` procedure 156
Figure 18.7: `initialize_storage` function 157
Figure 18.8: `load_storage` function 158
Figure 18.9: BASIC code for load_storage 158
Figure 18.10: `find_node` function 159

Figure 18.11: `update_qty` function 160
Figure 18.12: `update_price` function 161
Figure 18.13: Finding the last node in the linked list 163
Figure 18.14: Allocating a node from the free list 164
Figure 18.15: Appending the node with identifier 9 to the linked list 164
Figure 18.16: `append_node` function 166
Figure 18.17: Finding and removing node with identifier 0 from linked list . 169
Figure 18.18: Prepend deleted node to free list 170
Figure 18.19: Linked list and free list after delete 170
Figure 18.20: `delete_node` function 171
Figure 18.21: BASIC code for singly linked lists (1 of 5) 173
Figure 18.22: BASIC code for singly linked lists (2 of 5) 174
Figure 18.23: BASIC code for singly linked lists (3 of 5) 175
Figure 18.24: BASIC code for singly linked lists (4 of 5) 176
Figure 18.25: BASIC code for singly linked lists (5 of 5) 177
Figure A.1: Sample bas55 session 183
Figure E.1: Algorithm flowchart for converting °C to °F 206
Figure E.2: Minimal BASIC program for converting °C to °F 206
Figure E.3: If flowchart 207
Figure E.4: If flowchart for Minimal BASIC 207
Figure E.5: **`EX01.BAS`** 207
Figure E.6: **`EX01.BAS`** runtime output 207
Figure E.7: If with Else flowchart 208
Figure E.8: If with Else flowchart for Minimal BASIC 208
Figure E.9: **`EX02.BAS`** 208
Figure E.10: **`EX02.BAS`** runtime output 208
Figure E.11: Five-way branch flowchart 209
Figure E.12: **`EX03.BAS`** 209
Figure E.13: **`EX03.BAS`** runtime output 209
Figure E.14: While Loop 210
Figure E.15: While Loop for Minimal BASIC 210
Figure E.16: **`EX04.BAS`** 210
Figure E.17: **`EX04.BAS`** runtime output 211
Figure E.18: Do .. While loop 212
Figure E.19: **`EX05.bas`** 212
Figure E.20: **`EX05.BAS`** runtime output 212
Figure E.21: Repeat .. Until Loop 213
Figure E.22: Repeat .. Until Loop for Minimal BASIC 213
Figure E.23: **`EX06.BAS`** 213
Figure E.24: **`EX06.BAS`** runtime output 213

Figure E.25: Ascending Arithmetic For Loop 214
Figure E.26: **`EX07.BAS`** 214
Figure E.27: **`EX07.BAS`** runtime output 214
Figure E.28: Descending Arithmetic For Loop 215
Figure E.29: **`EX08.BAS`** 215
Figure E.30: **`EX08.BAS`** runtime output 215
Figure E.31: Subroutine 216
Figure E.32: Subroutine Call 216
Figure E.33: **`EX09.BAS`** 216
Figure E.34: **`EX09.BAS`** runtime output 216

List of Tables

Page

Table 5.1: Runtime Trace for Flowchart 4 26
Table 6.1: Factorial Runtime Trace With Input 5 41
Table 9.1: Column Skip Table 72
Table 15.1: How many iterations are required for a large binary search? . . . 128
Table 15.2: $O(n)$ table 128
Table 18.1: Singly linked list demo subroutine locations 172
Table E.1: Flowchart Symbols 205

Preface

This book is intended as a traditional introduction to programming using the original Dartmouth style of BASIC. In true old school style, it features algorithm development using simple flowcharts. While the combination of BASIC and flowcharts has fallen out of fashion, it is a proven method for teaching programming to both hobbyists and academic students. Admittedly it does require you to have a reasonable attention span, a good store of patience, and perseverance. These are the same attributes that are required to learn programming in any language.

The decline of the popularity of the BASIC language with hobbyists was accelerated by the decision by Microsoft® to cease bundling a free copy of BASIC with their operating systems. Another factor was the amazing Turbo Pascal® from Anders Hejlsberg[1] which was very fast, produced true executables, and included an IDE. It was also very cheap compared to other compilers, and many people switched from BASIC interpreters to compiled Pascal.

For educational use, the decline had a different cause. So many BASIC programs were written without obvious structure that the term *spaghetti BASIC* was coined to describe the especially bad style of BASIC code that used too many `GOTO` statements. The proliferation of BASIC programs written using awful style, and the difficulty of breaking the bad habits of that style of programming in students when they went on to learn other programming languages, led to Edsger W. Dijkstra's intensely bitter criticism of the language. As an acknowledged giant of computer science and a genuine Turing award winner, when Dijkstra said "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."[2], it carried incredible weight in the computer science field. His strong criticism helped eliminate the use of BASIC as a teaching language for academia.[3] Niklaus Wirth's Pascal language rigidly enforced what were considered better programming practices, and many schools used it as a replacement for BASIC in their teaching.

[1] Mr. Hejlsberg also designed Delphi and C#.

[2] "How do we tell truths that might hurt?" (EWD498), June 18, 1975, privately circulated.

[3] People tend to forget that in that same paper he also attacked FORTRAN, COBOL, the software engineering discipline, and even IBM mainframes. All of those technologies are still in use today, 40 years later.

For business, the decline of BASIC was at least partially due to the inability of ANSI to define and publish a reasonable standard. The Minimal BASIC standard did not include support for files, string operations, or arrays of strings and thus was too weak to be practical for business use. The Full BASIC standard was wildly ambitious, requiring among other things support for decimal math. It had far too many features to reasonably implement or even fit in the memory of the microcomputers of the time. To my knowledge, no complete implementation of Full BASIC was ever created for a personal computer before the standard was withdrawn. For business, Minimal BASIC was not enough, Full BASIC was too much, and so the language splintered into hundreds of not quite compatible dialects. This, combined with the tendency of BASIC programmers to write undocumented programs using spaghetti code, caused maintenance nightmares for businesses. They reacted by abandoning traditional BASIC altogether. While BASIC for business lives on in name with Visual BASIC® and TrueBASIC®, these are essentially Pascal derivatives and not really BASIC at all.

So if Minimal BASIC is too weak for business, why have I written both a compiler for the language and a book about it? BASIC was initially designed for teaching programming to non-technical people and worked well for that. Minimal BASIC is the simplest form of the language that was ever standardized, and provides enough features to solve many mathematical problems. Using a simple language with a small number of keywords and a small number of features reduces the amount of syntax and semantics you must memorize, and lets you spend more time actually programming instead. This BASIC dialect allows you to learn the essential concepts of procedural, iterative programming without getting lost in the advanced features of a modern, full-sized programming language. In short, using Minimal BASIC will keep things simple while still allowing you to program. I sincerely hope that after you master ECMA-55 Minimal BASIC, you will be motivated to continue your study by learning a more advanced programming language and the corresponding more advanced techniques such a language enables you to use. The vast majority of the problem solving skills you learn from this book will be valid for any mainstream programming language.

Much of the ugly style of early BASIC programs was a result of trying to get real programs to fit into the tiny amount of memory available to the interpreters at the time. Most early machines had far less than 64 kilobytes available for the interpreter, the loaded program, and the data the program used. Students tend to emulate code they know works, and much of the available working, non-trivial BASIC code was undocumented, re-used variables to save memory, used subroutines with multiple entry and exit points, and would be considered awful code today. For instance, in a constrained memory environment, comments were considered a waste of precious bytes. At the time it was written, the code that is considered to have truly awful style today was highly revered since it allowed packing more features

into less memory. While BASIC *permits* such code, it certainly does not require it. Even today, memory-saving tricks and clever control flow are considered to be essential for high-quality assembly code, but abominable for high-level code. BASIC programs do not *have* to be undocumented or inherently unstructured. With the large memory available in today's machines it is possible to write maintainable, structured BASIC programs with a current implementation of the classic language. This book aims to teach you step-by-step how to write maintainable programs in traditional BASIC without resorting to ugly style. The need for proper comments is stressed, and all of the complex example programs include appropriate comments.

The recommended way to use this text is to work through the chapters sequentially. In each chapter, when you read about a program, study the flowcharts, and then type in the example program and run it. Requiring you to actually type in the programs will build the typing and editing skills that are essential for you to achieve any real productivity in programming. It also forces you to actually read every line of code in the program. Once you have read the text, studied the flowchart, typed in a program, and established the one-to-one correspondence between the program and the corresponding flowchart, you should be able to understand both the algorithm and the ECMA-55 Minimal BASIC syntax required to implement that algorithm. It is expected that you may have to read chapters multiple times and experiment a bit. Exercises are provided at the end of each chapter to help reinforce the concepts from the chapter.

Two supported implementations of the ECMA-55 Minimal BASIC language exist today. Most students will choose Jorge Giner Cordero's excellent `bas55` interpreter since it is easy to use and runs on the popular Microsoft® Windows® platform as well as on most POSIX® systems. The `bas55` software is described in appendix A. For users with AMD64®-compatible systems running modern Linux®, I of course recommend my own `ecma55` compiler, which is described in appendix B. All examples in the book will work with either implementation. Two helpful quick reference guides are also provided. Appendix C is a guide to the keywords of the ECMA-55 Minimal BASIC language. Appendix D is a guide to the built-in functions of the ECMA-55 Minimal BASIC language.

Chapter 1

Introduction

Learning to program is easy using the ECMA-55 Minimal BASIC language. This dialect of BASIC is very similar to the original BASIC created by John George Kemeny and Thomas Eugene Kurtz at Dartmouth College in 1964. This was the first successful computer language invented for teaching, and it was created to allow absolute beginners who had no programming background at all to write and run their own computer programs.

In this manual, BASIC refers to the ECMA-55 Minimal BASIC standard dialect of BASIC. The first thing to know is that this language is line-based, with one statement on each line. A statement is an instruction to tell the computer to do something. In BASIC, each line must start with a unique unsigned integer value between 1 and 9999 called the *line number*. These numbers specify the order of the statements and give each statement a unique identifier. Here is a sample of a small program:

Program Source Code

```
10 REM DISPLAY ODD NUMBERS FROM 1 TO 9
20 LET I=1
30 PRINT I
40 LET I=I+2
50 IF I<=9 THEN 30
60 END
```

Here you will see there are six lines, each beginning with a line number, and the lines are displayed in ascending order by line number. When the program runs, it will start on the lowest numbered line. Unlike some other computer languages, tabs are not permitted in the source code, and each item on the line is separated by at least one space. The last line of the program must be an **END** statement, and no other line can have an **END** statement. Notice that each line has a statement that begins with a keyword after the initial line number. For example, the keyword on lines 20 and 40 is **LET**. Line 10 has the keyword **REM** which is used to enter a remark, or comment that does not do anything but is for the programmers to read. It is a way to put notes, or hints about how the program works, into the source

code of the program. BASIC requires that the `ASCII`[1] character set be used.[2] Lines can be up to 72 columns wide including the line number. Lower-case alphabetic characters are ***not*** permitted.

Now that you have seen a simple example program, the first question that should come to your mind is "What does the program tell the computer to do?" The best way to explain it is with a picture as shown in figure 1.1.

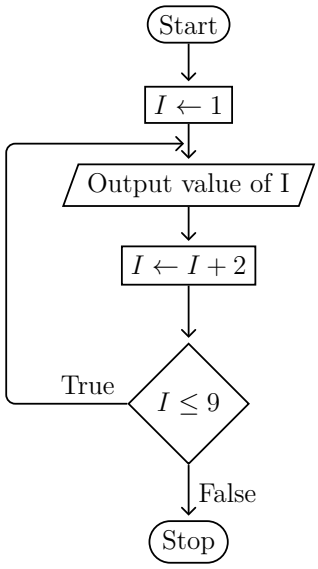


Figure 1.1: Sample Flowchart

The picture is called a *flowchart* and the arrows indicate the direction of the flow of logic. Flowcharts are independent of the programming language being used and represent the essence, or idea, of the program, which is known as an *algorithm*. This flowchart[3] shows many of the symbols that will be used in this document. The rectangle is used for assignment, which changes the value of variable. The parallelogram is used for statements that perform data input or output. The diamond is used for statements that have a condition that can change what the program will do. A special terminal symbol exists that looks like a rectangle that had $\frac{1}{2}$ circles glued on the left and right sides and it is used for the start of the program and the end of the program. This flowchart lets you see instantly that there is a loop in the program since if the condition in the diamond is true, the program will go back to an earlier place in the program and then continue going until it reaches the condition again. When using the flowchart, you start on the Start symbol and when you get to the terminal symbol with the word Stop, you quit running the algorithm. While there can only be one terminal symbol with the word Start, there can be multiple terminal symbols with the word Stop.

The second question that should have leapt into your mind when you first saw the example program was "What will it output?". There is a comment on line 10 that

[1] American Standard Code for Information Interchange

[2] Actually, the standard specifies the ECMA-6 7-Bit Coded Character Set. This was an ECMA-ratified version of ISO/IEC 646. The modern version of the international standard is ISO 646:1991, also known as ITU T.50. The ISO 646:1991 IRV (International Reference Version) is identical to ASCII. Thus current implementations of ECMA-55 Minimal BASIC uses (part of) the ASCII character set since ECMA-6 is a subset of ASCII. BASIC needs the IRV version for the DOLLAR SIGN at 2/4 and the CIRCUMFLEX ACCENT at 5/14.

[3] The flowcharts in this manual comply with the program flow charts specified in the "STANDARD ECMA-4 FLOW CHARTS, 2nd Edition, September 1966" document.

says this will output odd numbers between 1 and 9, but will it really? Here is the output from that program:

Program Output

```
1
3
5
7
9
```

As you can see in the program output, the program generates the odd numbers between 1 and 9, printing each of those odd numbers on its own separate line, so the comment is correct. So how is the BASIC program really an implementation of the algorithm specified by the flowchart? In BASIC, comments use the **REM** keyword which is short for remark. Everything after the **REM** is ignored, as long as no lower-case alphabetic characters are used. The comments, or **REM**arks specified by the **REM** statement are not included in flowcharts, since they do not actually make the program do anything. The rectangles in the flowchart represent **LET** statements which are used to assign values to variables. The values may be constants or the results of expressions. The parallelogram is used for the **PRINT** statement which displays information. The diamond corresponds to the **IF** statement which is used to do a conditional branch. The Stop picture is how the **END** statement is drawn. The Start picture does not have a special statement in BASIC, but instead corresponds to the first non-**REM** statement in the program.

Chapter 2

HELLO, WORLD!

Now it is time to begin programming. Traditionally, a "Hello, World!" program is the first program a student writes. The required logic is very simple, but even a simple program has a corresponding flowchart. The flowchart for the "HELLO, WORLD" program is shown in figure 2.1.

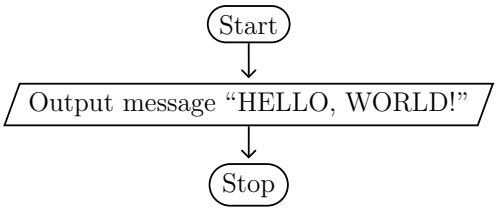


Figure 2.1: Hello World Flowchart

This resulting program is called a *straight-line* program because the execution flow through the flowchart is in a straight line from the start symbol to the stop symbol and there are no branches. A *branch* occurs when after a statement is executed the program continues running at some location besides the line of the program immediately following that statement. In the introduction you already learned every program must have an **END** statement as the last line, and when you want to display something on the screen we use the **PRINT** statement. The source code for the program implementing the *algorithm* in flowchart 2.1 in the ECMA-55 Minimal BASIC dialect is shown in figure 2.2.

```
10 PRINT "HELLO, WORLD!"
20 END
```

Figure 2.2: Hello World Program

What does line 10 do? The **PRINT** statement will send the message `HELLO, WORLD!` to your screen. Then the program will advance to the next line, line 20. Line 20 is the special **END** statement which marks the end of the program, so the program will terminate. The output of the program when you run it is shown in figure 2.3.

```
HELLO, WORLD!
```

Figure 2.3: Hello World Program Output

Line numbers are traditionally entered starting with 10, not 1, and in increments of 10 instead of increments of 1. This is to allow adding more lines later to modify the program. This leaves room for 9 lines before the first line, and room for 9 lines between any other two lines in the program.

Exercises

1. Create a flowchart for an algorithm that displays the message "I CAN PROGRAM!"
2. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

Chapter 3

Temperature Conversion

The easiest problems that can be solved with Minimal BASIC are math problems that are based on a formula. Conversions of various units of measurements are typical examples of problems that can be solved by using a simple formula. This chapter will work through two examples of temperature conversion to show you step-by-step how to work from a formula to a flowchart and finally to a working Minimal BASIC program.

3.1 Convert Celsius to Fahrenheit

When you travel, you may want to be able to convert the temperature in Celsius to the temperature in Fahrenheit. The formula for this is well known, and is shown in equation 3.1.

$$^\circ F = 32 + \frac{9}{5}{}^\circ C \tag{3.1}$$

The next step after you have the formula is to create a flowchart that uses that formula. The flowchart will describe the logic we need, called the *algorithm*. The algorithm does not specify what computer language will be used, only what the program must do, step-by-step. A suitable flowchart is shown in figure 3.1.

To get the value of the temperature in Celsius from the user we need to use the `INPUT` statement which gets information from the keyboard. The `INPUT` keyword is followed by a comma-delimited[1] list of one or more variable names. Since our formula and flowchart both use C and F, it makes sense to use those as the variable names in our program. You need to get the value of C from the user, and you can do that by using `INPUT C`. In any flowchart the `INPUT` statement, like the `PRINT` statement, is always shown as a parallelogram.

A BASIC program that implements flowchart 3.1 to convert a Celsius value typed in by the user to a Fahrenheit value using formula 3.1 is shown in figure 3.2.

Once you have typed in the program, it is time to try running it. First try converting 0°C to Fahrenheit. We know it should be 32, since this is the temperature at which water normally freezes. You can see the output of the program in figure 3.3.

[1] A delimiter is something that separates items in a list. So comma-delimited items are a list of items with commas separating them.

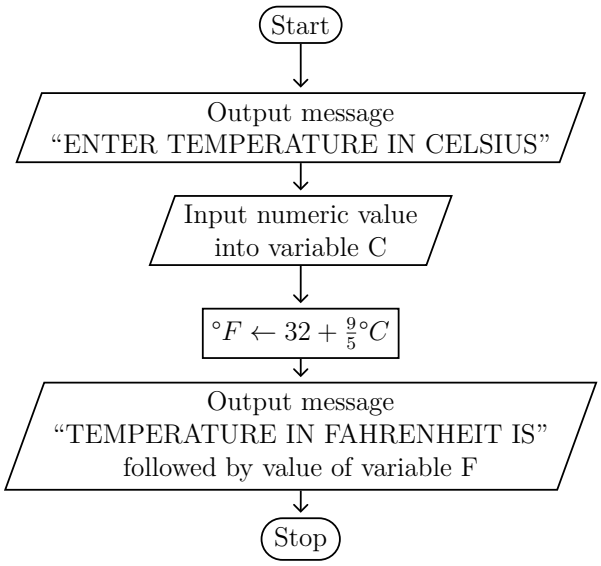


Figure 3.1: Flowchart for Converting °C to °F

```
10 PRINT "ENTER TEMPERATURE IN CELSIUS";
20 INPUT C
30 LET F=32+(9*C)/5
40 PRINT "TEMPERATURE IN FAHRENHEIT IS";F
50 END
```

Figure 3.2: Program Celsius to Fahrenheit

```
ENTER TEMPERATURE IN CELSIUS? 0
TEMPERATURE IN FAHRENHEIT IS 32
```

Figure 3.3: Program Celsius to Fahrenheit Output for 0°C

On a really beastly hot day in Bangkok, Thailand, the temperature might be as high as 43°C which should be just over 109°F. When we run the program with that input you will see in figure 3.4 that indeed the program works and gets the correct answer.

```
ENTER TEMPERATURE IN CELSIUS? 43
TEMPERATURE IN FAHRENHEIT IS 109.4
```

Figure 3.4: Program Celsius to Fahrenheit Output for 43°C

Now you have successfully written programs that can use a formula for calculation. Amazingly, well into the 1960's almost everyone wanting to do those conversions had to use a pencil and paper and do it by hand. Today almost all mobile telephones

have a calculator that can do it. In this chapter you have learned a lot already. Here is a list you can use to help you review what you have learned. The list shows what you know about the flowchart symbols in figure 3.5 for the following kinds of actions:

- *Beginning a program*
  The program will always begin on the lowest numbered line of the program. This uses the terminal symbol.
- *Displaying output on the screen*
  This requires using the **`PRINT`** statement. This uses the input/output symbol.
- *Getting input from the keyboard*
  This requires using the **`INPUT`** statement. This uses the input/output symbol.
- *Assigning a value to a variable*
  The value can be a literal value or the result of an arithmetic expression. This requires using the **`LET`** statement. This uses the rectangle symbol.
- *Ending a program*
  This requires using the **`END`** statement. This uses the terminal symbol.

Figure 3.5: Flowchart Symbols

3.2 Convert Fahrenheit to Celsius

It's time for you to try another conversion program, but this time you will write a program to convert from Fahrenheit to Celsius. The correct formula for that is given in equation 3.2.

$$°C = \frac{5}{9}(°F - 32) \tag{3.2}$$

Once you have the formula and think about it a while, you will realize the flowchart will look almost the same as the one for converting from C to F. A flowchart to convert from Fahrenheit to Celsius is shown in figure 3.6 on page 12.

Since the flowchart shown in figure 3.6 is very similar to the previous program's flowchart shown in figure 3.1 on page 10, it is obvious that the program for this flowchart will be similar to the previous program. Figure 3.7 on page 12 shows a BASIC program that uses the algorithm specified by the flowchart to convert a Fahrenheit value typed in by the user to a Celsius value using equation 3.2.

Well, the good news is that the conversion is correct. The bad news is that as you can see in figure 3.8 on page 12, there is no space between the `S` and the minus sign. What happens for answers that are positive?

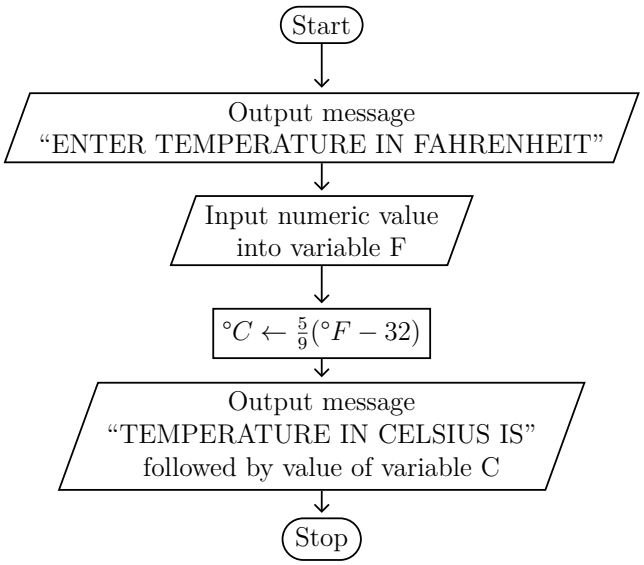


Figure 3.6: Flowchart for Converting °F to °C

```
10 PRINT "ENTER TEMPERATURE IN FAHRENHEIT";
20 INPUT F
30 LET C=(5*(F-32))/9
40 PRINT "TEMPERATURE IN CELSIUS IS";C
50 END
```

Figure 3.7: Program Fahrenheit to Celsius

```
ENTER TEMPERATURE IN FAHRENHEIT? 0
TEMPERATURE IN CELSIUS IS-17.7778
```

Figure 3.8: Program Fahrenheit to Celsius Output for 0°F

```
ENTER TEMPERATURE IN FAHRENHEIT? 43
TEMPERATURE IN CELSIUS IS 6.11111
```

Figure 3.9: Program Fahrenheit to Celsius Output for 43°F

As you can see in figure 3.9 on page 12, positive values work without a problem. The error in our program only occurs if the answer is negative. We will learn the necessary techniques to handle this case in later chapters. Some programmers will call an error a *bug*, and that is the basis of the words *debug* (remove a bug) and *debugger* (a tool to help find bugs so you can remove them).

Note that the actual output for floating point values may be different in your implementation of ECMA-55 Minimal BASIC than the values shown here. The ECMA-55 standard allows the implementation to determine some of the floating point details. For instance, the number of digits after the period in 1/3 may differ between implementations, but will be something like 3.33334E-1 or .3333334, but the number of '3' digits after the period and whether or not to use scientific notation will depend on the implementation. This issue is not unique to ECMA-55 Minimal BASIC. For instance, even programs written in the famous C computer language can have math results that differ for the same program depending on the compiler and the platform used.

Exercises

1. Create a flowchart for an algorithm that converts from inches to centimeters. One inch is equal to 2.54 centimeters.
2. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.
3. Create a flowchart for an algorithm that converts an angle in degrees to an angle in radians. 360° is equal to 2π radians.
4. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

Chapter 4

Scalar Variables and Constants

In chapter 3 programs were written that used variables. What are variables? A *variable* is a name for a region of memory in your program's address space. In simple terms, it's a memory address. Your program runs in a process. That process uses memory, and in most machines today that is RAM (Random Access Memory). Memory is organized into words, and those words are divided into bytes, and bytes are collections of 8 bits, and a bit is a single part of memory that can be 0 or 1. Once upon a time, long long ago, programs were written directly in machine code without using variable names and instead the numeric addresses were used. That is hard for humans to remember, and made reading programs very difficult. Today instead of having to remember the address of every place in memory where we will store data, we use variable names, normally just called variables.

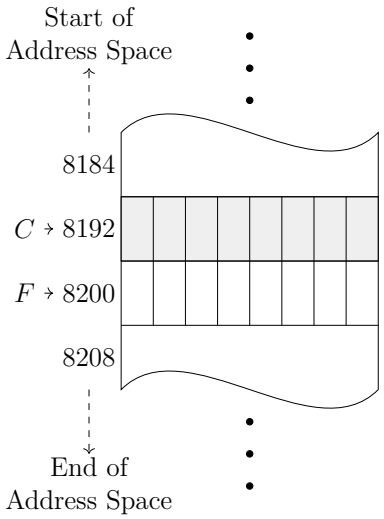


Figure 4.1: Scalar Variables are Addresses

For example, in the program shown in figure 3.7 on page 12, two scalar numeric variables are used: C and F. Conceptually, these are in memory as shown in figure 4.1. The variable C is actually stored at address 8192, and the variable F

is after that at address 8200.[1] Each scalar will store a number, and in Minimal BASIC on a 64 bit machine, each number is a floating point value and typically uses 8 bytes. In languages like C and C++, the data type used is called a *double*.

The good news for you is that when you program in BASIC, you do not need to think about how many bytes or what the addresses are where the data is stored. All you need to do is use variables and the computer will automatically find memory and track the address where each variable will be stored. In ECMA-55 Minimal BASIC, you get 26 scalar numeric variables with names from A to Z, and then you get 260 more with names from $A0$ to $Z0$, $A1$ to $Z1$,...,$A9$ to $Z9$. What do I mean by a *scalar* variable? A scalar variable is a variable that can hold just one unit of data. In ECMA-55 Minimal BASIC, numeric data is always stored as doubles, so a numeric scalar variable in BASIC can hold one double type number. A *double* is a *floating point* number, which is a computer approximation of a *real number*. You already know what a *real number* is, and that some of them such as π and e have an infinite number of digits. A computer cannot store every real *real number* perfectly because storage is limited and it would not be efficient to do math with unbounded-sized numeric representations. Floating point numbers approximate real numbers, but because the storage for each number is limited, and thus the precision is limited, the computer cannot natively store all real numbers. To do that, we would need infinite room, since the number of real numbers, even between any two real numbers, is infinite.

The IEEE754[2] standard supports a double type which all known modern implementations of ECMA-55 Minimal BASIC use for numeric data. This is a floating point type for numbers with about 15 decimal digits in the mantissa, and 3 decimal digits in the exponent, although the number is actually stored in a *binary* (base-2) format. If you try to use numbers with more than 15 decimal digits of mantissa, the excess precision will be truncated to make the number fit within the allocated bytes of storage. On a 64 bit machine with a 64 bit implementation of ECMA-55 Minimal BASIC, 8 bytes of storage will be allocated. If you use numbers with too large an exponent, an overflow error will occur. The range of the positive floating point numbers for AMD64 machines is approximately from 2.2×10^{-308} to 1.79×10^{308}, and the range of the negative floating point numbers is approximately from -1.79×10^{308} to -2.2×10^{-308}.

Sometimes you want to store letters instead of numbers. A stream of characters is called a *string*, and is enclosed in double quotes when you type a literal value. In BASIC, a string scalar variable has a trailing dollar sign and there are 26 variables

[1] The actual addresses and bit patterns used to store values depend on the platform, but those details are taken care of automatically for the programmer by their ECMA-55 Minimal BASIC implementation.

[2] IEEE 754-1985 as it is implemented in Intel64® and AMD64® 64 bit processors.

you can use, *A*$ through *Z*$. In ECMA-55 Minimal BASIC a string can only contain a maximum of 18 characters, and they must be 7 bit ASCII characters. Lower-case characters are not permitted.[3] A string scalar variable, like a numeric scalar variable, is really just an alias for the address of the location in memory where the string is stored.

A *data type*, sometimes shortened to just *type*, is a term that means the kind of information that can be stored in a variable. ECMA-55 Minimal BASIC has just two scalar types: numeric and string. If a variable name has a trailing dollar sign then the type is string, otherwise it is numeric. You *must* put a value into a variable, usually with a **LET** statement, before you try to use the variable in any expression. Unlike many modern languages, you do not need to declare scalar variable names or their types with ECMA-55 Minimal BASIC. The variable name itself specifies the type, and storage for variables is allocated automatically by the BASIC software when you use them in your program. Since the variable names are so short, for any long program you should include some comments using the **REM** statement to describe the purpose of each variable.

```
10 REM TEMPERATURE CONVERSION FROM F TO C
20 REM F IS THE TEMPERATURE IN FAHRENHEIT THE USER WILL SUPPLY
30 REM C IS THE TEMPERATURE IN CELSIUS THE PROGRAM WILL COMPUTE
40 REM
50 PRINT "ENTER TEMPERATURE IN FAHRENHEIT";
60 INPUT F
70 LET C=(5*(F-32))/9
80 PRINT "TEMPERATURE IN CELSIUS IS";C
90 END
```

Figure 4.2: Program Fahrenheit to Celsius with Comments

In the sample program shown in figure 4.2 a comment using a **REM** statement is used to describe what the program does on line 10. Then on lines 20 and 30 comments exist that describe the purpose of the variables *F* and *C*. Line 70 is the formula, and uses three *literal values*, the values 5, 32, and 9. Line 70 will compute the expression, which is everything after the equals sign, and then store the resulting value in the variable name specified before the equals sign. This is called assignment of a value to a variable, and the **LET** statement is used for that. Line 60 also assigns a value to a variable, but it gets the value from the keyboard because it is an **INPUT** statement.

So why are these names of places to store data called variables? That name comes from mathematics, where a *variable* is the name of a numeric value that can vary,

[3]ECMA-116 Full BASIC has better string support that allows more characters, substring addressing, concatenation, and many other things, but ECMA-55 Minimal BASIC is designed for math problems and lacks the advanced string features.

or change, over time or with different conditions. A value that cannot vary is called a *constant*. In programming with BASIC, a constant is often called a *literal value*, sometimes shortened to just a *literal*.

To review, there are four types of data ECMA-55 Minimal BASIC can use that have been discussed in this chapter:

1. `"HELLO"` is a string literal, or a scalar string constant.
2. `99` is a numeric literal, or a scalar numeric constant. Many different formats of numeric constants exist, and some examples will let you know what to expect:
   - `3.1415926` (π)
   - `314.15926E-02` (π)
   - `-.31415926E+01` ($-\pi$)
   - `2.7182818` (e)
3. `A$` is a string scalar variable.
4. `A` and `Z4` are numeric variables.

Exercises

1. Describe the difference between a numeric scalar variable and a numeric constant in ECMA-55 Minimal BASIC.

2. Describe the difference between a string scalar variable and a string constant in ECMA-55 Minimal BASIC.

3. Describe the difference between a numeric scalar variable and a string scalar variable in ECMA-55 Minimal BASIC.

Chapter 5

Generating Sequences

This chapter contains several examples of programs that generate sequences of numbers. We will begin with a straight-line program, and then gradually improve it. As part of writing the improvements, we will learn about more BASIC statements. But first, you need to understand sequences. Actually, you probably already know about them, but this quick review will help you if you forgot everything. The simplest sequence most people know is the sequence of counting numbers that goes like this:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots$$

The trailing ... indicates that the sequence goes on forever. Probably you do not want to run your program forever, so the number of terms to be printed in a sequence must be limited. There are two common ways to limit the number of terms printed in sequences:

- Stop after a specified number of terms have been printed.
  For example, if we want five terms of the counting number sequence, we would get 1,2,3,4,5 and then stop.
- Stop after a specified threshold value has been reached.
  For example, if we want to stop when we exceed the value 7, we would get 1,2,3,4,5,6,7 and then stop.

Now it is time to specify exactly what we want, create the flowchart, then write the program and finally run it. For this first program, we will stop after we print five terms of the counting numbers sequence.

You can see the shape of the flowchart in figure 5.1 on page 22 is exactly the same as the flowchart for the program that printed "HELLO, WORLD!". The only difference is the message printed this time is "1,2,3,4,5". The corresponding BASIC program is shown in figure 5.2 on page 22. When the program in figure 5.2 is run, the output of the program is correct, as shown in figure 5.3 on page 22.

In this program I included lines 10 and 20 with some comments using the `REM` statement. Those lines describe what the program is doing. It is a good idea to

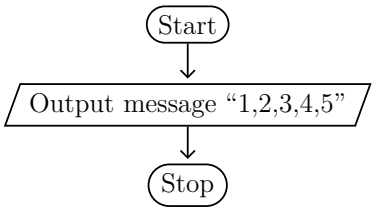


Figure 5.1: Flowchart 1

```
10 REM PROGRAM TO PRINT FIRST FIVE VALUES IN THE
20 REM COUNTING NUMBER SEQUENCE
30 PRINT "1,2,3,4,5"
40 END
```

Figure 5.2: Program 1

```
1,2,3,4,5
```

Figure 5.3: Program 1 output

include the comments because in a week or two you may forget what the program was doing. If you have no comments, it can take a while to learn what a program is trying to do. Comments are not included in a flowchart, only in the source of a program.

Well, that program was easy and works, but what happens if we want to ask the user for the number of terms and then print those terms? Well, we know we can get the number of terms using the `INPUT` statement. If we are on a term T, we can use formula shown in equation 5.1 to get the next term:

$$T = T + 1 \tag{5.1}$$

But how can we know when to stop? How can we avoid typing that formula many, many times? We will need to learn some new BASIC statements to make it possible to write that program. But first, we need to create a flowchart! In the introduction the diamond symbol was used to allow checking a condition and then going one way if the condition was true and another way if the condition was false. When the program flow goes to different parts of the program depending on a condition this is called a *conditional branch*. That is part of what we need to solve this problem. We will need two variables. One will count how many terms we have printed, and the other will hold the value of the current term to print. Study the flowchart shown in figure 5.4 on page 23.

At first glance the flowchart looks like it will be fine, and for the sequence we are printing it will work. However, what if we want a sequence like the positive even

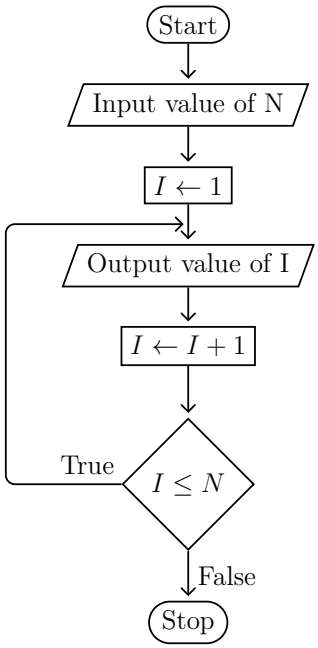


Figure 5.4: Flowchart 2

numbers? That sequence is:

$$2, 4, 6, 8, 10, \ldots$$

Changing the formula for computing I and the starting point are the first steps, which result in the flowchart show in figure 5.5 on page 24. If we run that flowchart by hand, however, and enter 5, we only get this sequence:

$$2, 4$$

Running a flowchart is pretty easy. You need to create some place to record the values of the variables and then update them when they are updated in the flowchart. You need to keep track of your position in the flowchart as you run the algorithm. For the flowchart in figure 5.5 on page 24, we need two variable locations, one for the variable I and another for the variable N. Let's run the flowchart step-by-step. We begin at the start symbol. The we enter a 5 when prompted so the value of N becomes 5. The variable I is assigned the value of 2. Then we print the 2. Next we increment the variable I by 2 and the value of I becomes 4. We now check whether the value of I, now 4, is less than or equal to the value of N, now 5. The answer is true, so we go back and print the new value of I which is 4. Next we increment I by 2 and the value of I becomes 6. We now check whether the value of I, now 6, is less than or equal to the value of N, now 5. The answer is false, so the program continues to the end symbol and execution of the algorithm ends after printing only two terms of the series instead of five terms like we wanted.

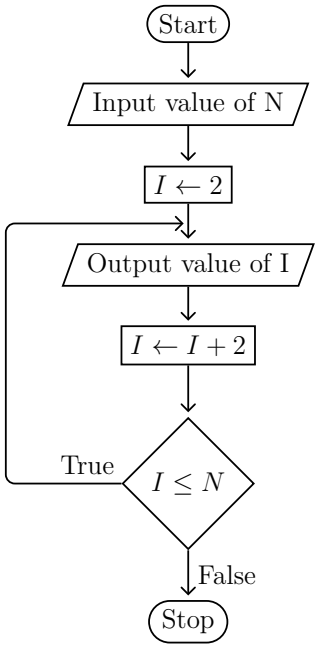


Figure 5.5: Flowchart 3

This bug occurs because we really need to have separate variables for the count and the next sequence value. If C is used for the count variable, and a different variable I is used for the current term variable, the problem can be solved. A corrected flowchart with those changes is shown in figure 5.6 on page 25.

5.1 Loops

Notice that in the flowcharts shown in figure 5.5 on page 24 and figure 5.6 on page 25 the arrows can go in a circle if the condition in the diamond is true. That circle, or cycle, is called a *loop*, and loops are one of the fundamental building blocks of algorithms. When you are executing an algorithm with a flowchart, or when you are running a real program, each time you go into the loop is called an *iteration* of the loop. In the examples so far in this chapter, the test to determine whether to do another loop iteration is always at the end of the loop. This type of looping structure is called a *post-test* loop. In a post-test loop, the test to determine whether to execute the loop body or exit the loop occurs at the end of the loop, after the loop body.

What happens when you run the corrected flowchart? Well, first you need to have memory locations for the three variables N, I, and C. Then you must determine what the user's input will be. In this case, the input will be 5. Running the algorithm with that value as input produces a table similar to table 5.1 which begins on page 25. It shows the changes of the variables while the algorithm runs.

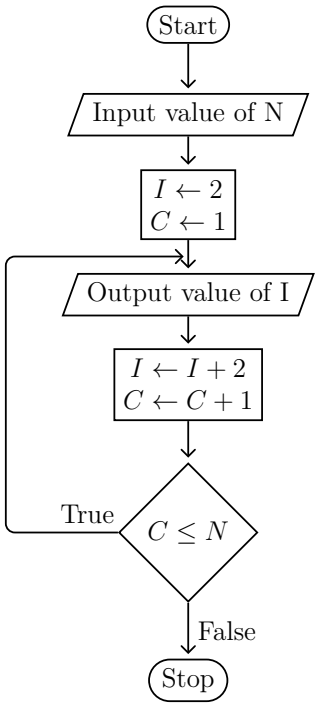


Figure 5.6: Flowchart 4

Iteration	C	I	N	Comments
	?	?	?	At the start all values are undefined
	?	?	5	$N \leftarrow 5$ from user input
	?	2	5	$I \leftarrow 2$
	1	2	5	$C \leftarrow 1$
1	1	2	5	display value of I which is 2
1	1	4	5	$I \leftarrow I + 2$
1	2	4	5	$C \leftarrow C + 1$
1	2	4	5	loop again because the value of C, which is 2, is less than or equal to the value of N, which is 5
2	2	4	5	display value of I which is 4
2	2	6	5	$I \leftarrow I + 2$
2	3	6	5	$C \leftarrow C + 1$
2	3	6	5	loop again because the value of C, which is 3, is less than or equal to the value of N, which is 5
3	3	6	5	display value of I which is 6
3	3	8	5	$I \leftarrow I + 2$

Continued on next page

Iteration	C	I	N	Comments
Continued from previous page				
3	4	8	5	$C \leftarrow C + 1$
3	4	8	5	loop again because the value of C, which is 4, is less than or equal to the value of N, which is 5
4	4	8	5	display value of I which is 8
4	4	10	5	$I \leftarrow I + 2$
4	5	10	5	$C \leftarrow C + 1$
4	5	10	5	loop again because the value of C, which is 5, is less than or equal to the value of N, which is 5
5	5	10	5	display value of I which is 10
5	5	12	5	$I \leftarrow I + 2$
5	6	12	5	$C \leftarrow C + 1$
5	6	12	5	exit because the value of C, which is 6, is **NOT** less than or equal to the value of N, which is 5

Table 5.1: Runtime Trace for Flowchart 4

The flowchart shown in figure 5.6 on page 25, and especially the runtime trace shown in table 5.1 which begins on page 25 from running that flowchart, make it clear that we can use the flowchart shown in figure 5.6 on page 25 to generate either sequence we considered with only a small change. Notice that in the first sequence I is always incremented by 1, but in the second case I is always incremented by 2. Otherwise the same flowchart would generate either sequence. What if we wanted to generate this sequence?

$$10, 20, 30, 40, 50, 60, 70, 80, 90, 100, \ldots$$

To generate this sequence you would just increment I by 10 instead of 1 or 2. You also need to have the user enter the first term value. So the increment should probably be a new variable which will be called D. Making those changes will result in the flowchart shown in figure 5.7 on page 27.

The flowchart shown in figure 5.7 on page 27 was changed in only two places, one to have the user input D as well as N, and the other to increment by D instead of 2. So what is the BASIC program for this flowchart? The most direct translation is shown in figure 5.8 on page 27.

Two things are new in this program. First, the `INPUT` statement on line 80 now has a list of variables instead of just one. The user will see a prompt, and then the user must enter 3 values with commas separating them. This is just an extended

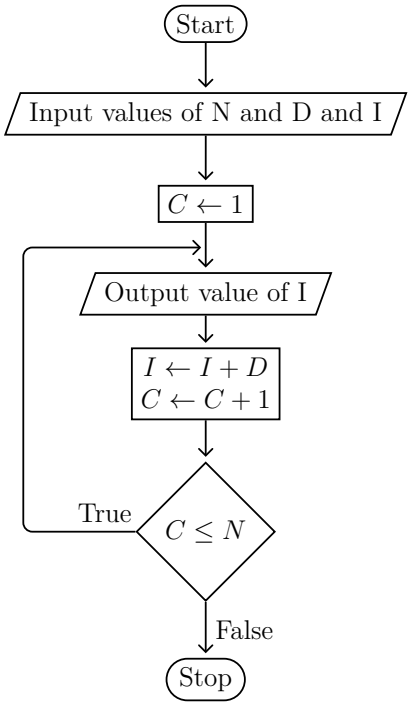


Figure 5.7: Flowchart 5

```
10 REM SEQUENCE GENERATING PROGRAM
20 REM D IS THE DELTA, OR AMOUNT TO ADD TO THE OLD VALUE TO GET THE NEXT
30 REM   VALUE IN THE SEQUENCE
40 REM N IS THE NUMBER OF TERMS WE WANT
50 REM I IS THE CURRENT TERM VALUE
60 REM C IS THE COUNTER FOR NUMBER OF TERMS PRINTED
70 PRINT "WHAT VALUES FOR NUMBER OF TERMS,DELTA,FIRST TERM";
80 INPUT N,D,I
90 LET C=1
100 PRINT I
110 LET I=I+D
120 LET C=C+1
130 IF C<=N THEN 100
140 END
```

Figure 5.8: Program Source Version 1

syntax for the **INPUT** statement you already know about. Line 130 has an **IF** statement, which is new for you. The **IF** statement is a conditional branch. The text between the **IF** keyword and the **THEN** keyword is the condition to check. If it is false, nothing happens and the program would continue on to the next line, in this case line 140. If the condition is true, then the program will branch, or jump, to the line specified after the **THEN** keyword, which in this case is line 100. This **IF** statement directly corresponds to the diamond in the last flowchart. The output of the program when you run it would be as shown in figure 5.9.

```
WHAT VALUES FOR NUMBER OF TERMS,DELTA,FIRST TERM? 5,10,10
 10
 20
 30
 40
 50
```

Figure 5.9: Program Output

What if the user replies with a value of 0 for N? The program should exit without emitting any numbers in the sequence, but because the test is **after** the output, the program will always show at least one term, which is an error. Several ways exist to fix this. Perhaps the first one you might think of is to add another test after the user provides input but before the loop begins.

Your first attempt to fix this bug would probably result in a flowchart that looks similar to figure 5.10. This algorithm does avoid printing a term if the user specifies zero terms. However, the algorithm is now more complex since it has two branches. You know two conditional branches exist because there are two diamonds in the flowchart. This will mean that the program will be more complex too. The BASIC program, after updating it to match the flowchart shown in figure 5.10 on page 29, is shown in figure 5.11 on page 29.

Is it possible to solve the problem with only one branch? In fact it is, and that solution is better than this one. Sometimes a solution works, but it is not the best solution. When possible, you will want to take the time to find the best solution since that solution will be easier to understand and update later.

If we moved the check from the diamond at the bottom so that the check is done first, and then printing and incrementing the variables occurs if and only if the check passes, then we would not need two diamonds.

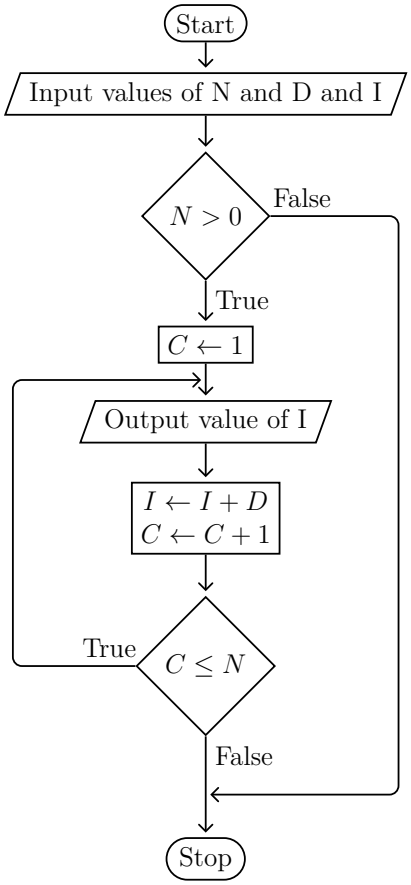


Figure 5.10: Flowchart 6

```
10 REM SEQUENCE GENERATING PROGRAM
20 REM D IS THE DELTA, OR AMOUNT TO ADD TO THE OLD VALUE TO GET THE NEXT
30 REM   VALUE IN THE SEQUENCE
40 REM N IS THE NUMBER OF TERMS WE WANT
50 REM I IS THE CURRENT TERM VALUE
60 REM C IS THE COUNTER FOR NUMBER OF TERMS PRINTED
70 PRINT "WHAT VALUES FOR NUMBER OF TERMS,DELTA,FIRST TERM";
80 INPUT N,D,I
90 IF N<1 THEN 150
100 LET C=1
110 PRINT I
120 LET I=I+D
130 LET C=C+1
140 IF C<=N THEN 110
150 END
```

Figure 5.11: Program Source Version 2

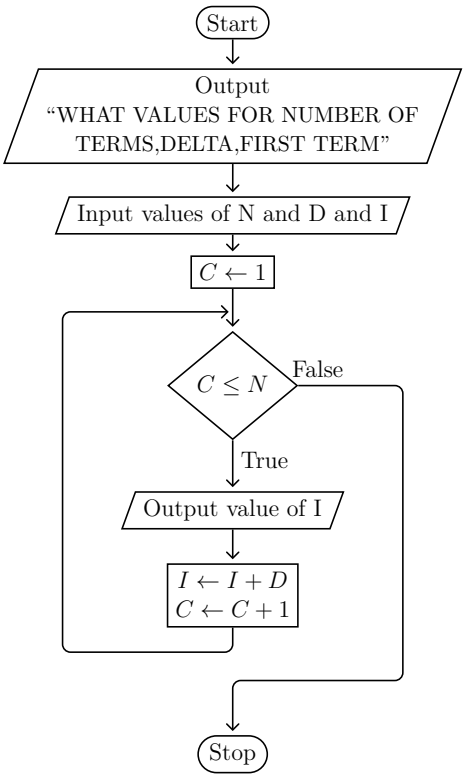


Figure 5.12: Flowchart 7

In figure 5.12 on page 30, you will see that there is a line that goes in a loop but it leaves from a rectangle, not a diamond. That means that we must make an *unconditional branch*, sometimes called a *jump*. You will need to use the **GOTO** keyword for that. The **GOTO** keyword is followed by a line number, just like with the **THEN** in the **IF** *conditional* branch statement. The line number specified after the **GOTO** or **THEN** keyword is called the *jump target*, and specifies the line number of the program where the control flow will go if the branch is taken. For **GOTO**, the branch is of course always taken. In the program for the flowchart shown in figure 5.12 on page 30, line 140 uses the **GOTO** keyword to implement an unconditional branch as shown in figure 5.13.

```
10 REM SEQUENCE GENERATING PROGRAM
20 REM D IS THE DELTA, OR AMOUNT TO ADD TO THE OLD VALUE TO GET THE NEXT
30 REM   VALUE IN THE SEQUENCE
40 REM N IS THE NUMBER OF TERMS WE WANT
50 REM I IS THE CURRENT TERM VALUE
60 REM C IS THE COUNTER FOR NUMBER OF TERMS PRINTED
70 PRINT "WHAT VALUES FOR NUMBER OF TERMS,DELTA,FIRST TERM";
80 INPUT N,D,I
90 LET C=1
100 IF C>N THEN 150
110 PRINT I
120 LET I=I+D
130 LET C=C+1
140 GOTO 100
150 END
```

Figure 5.13: Program Source Version 3

5.2 FOR Loops

Loops with the diamond at the top of the loop are called *pre-test* loops. For pre-test loops like the one shown in figure 5.12 on page 30, we can do even better. BASIC has a special pair of statements to support a nicer syntax for pre-test loops that count by an increment, The start of the loop must declare the variable that will be used for counting, the start value, and the end value. That special variable used for counting is known as the *loop index variable*, often shortened to just *loop index*. The default increment is one, but you can change the increment value when necessary. The statement used at the start of the loop is **FOR**. Line 140 in the source program version 3 has the end of the loop on line 140. Immediately before that on line 130 the loop counter variable C was incremented by one. The statement to do these two things is the **NEXT** statement. Every **FOR** must have a corresponding **NEXT**. Learning this seems complicated, but it really is just an easier syntax for pre-test counting loops.

Version 4 of the program is shown in figure 5.14 on page 32. There is no new flowchart since the algorithm does not change. The **FOR** statement on line 90

declares the variable **C** as the loop index variable, initializes it to the value **1**, sets the limit expression to the variable **N**, and set the increment to be **1**. This is the increment for the **C** variable that counts the number of times through the loop, not the increment for the sequence which is the variable **D**. So the lines 80 and 100 in version 3 are combined into line 90 in version 4.

The **TO** separates the initialization of the loop index variable C from the limit of the loop. The optional **STEP** clause specifies the increment to use. If no **STEP** is specified, then the increment used will be the default increment of 1. The end of the loop must increment the loop index variable, and then it must branch back to the test (diamond in the flowchart) at the top of the loop. This combines lines 130 and 140 in version 3 and replaces them by the one line with the **NEXT** statement on line 120 in version 4. With version 3, we needed to specify two line numbers, one for the target of the conditional branch, and one for the target of the unconditional branch. When using the **FOR**...**NEXT** looping structure you do not need to do that. So what happens when the condition to continue the loop is false? The code will branch to the line immediately following the **NEXT** statement. In version 4 of the program, when the loop exits the program will jump to line number 130.

```
10 REM SEQUENCE GENERATING PROGRAM
20 REM D IS THE DELTA, OR AMOUNT TO ADD TO THE OLD VALUE TO GET THE NEXT
30 REM   VALUE IN THE SEQUENCE
40 REM N IS THE NUMBER OF TERMS WE WANT
50 REM I IS THE CURRENT TERM VALUE
60 REM C IS THE COUNTER FOR NUMBER OF TERMS PRINTED
70 PRINT "WHAT VALUES FOR NUMBER OF TERMS,DELTA,FIRST TERM";
80 INPUT N,D,I
90 FOR C=1 TO N STEP 1
100 PRINT I
110 LET I=I+D
120 NEXT C
130 END
```

Figure 5.14: Program Source Version 4

The output of version 4 is identical to the output shown before in figure 5.9. There are some more details to remember about **FOR** loops. Each time the **NEXT** statement is reached, the loop index is incremented by a constant value. This value defaults to one, but you can specify the value you want to increment by with the optional **STEP** clause of the **FOR** statement. You can omit the "**STEP** 1" on line 90 if you want to avoid typing so much, since the default increment is already 1. Also, the **NEXT** that ends the loop must specify the same index variable as the **FOR** that starts the loop. The part of the loop between the **FOR** and **NEXT** statement is called the *body* of the loop. As you already know by now, describing the exact details of the **FOR** and **NEXT** statements takes many lines of text, but the idea is easy and makes writing counting loops, which are in almost every complex algorithm, much

easier to write. You can use a negative value in the **STEP** clause if you want to count down instead of count up, but in that case be careful to ensure the limit you specify is not actually greater than your start value. When the **FOR** loop exits, remember that the index variable will not be the last value of the index variable used. Instead, it will be the value that would have been used in the next iteration of the loop. That is, it will be the sum of the last loop index variable value used and the step value. Type in the program shown in figure 5.15 and run it. You will see the output shown in figure 5.16.

```
10 FOR X=1 TO 9 STEP 4
20 PRINT X
30 NEXT X
40 PRINT "AFTER LOOP X IS";X
50 END
```

Figure 5.15: Program Source Version 5

```
 1
 5
 9
AFTER LOOP X IS 13
```

Figure 5.16: Output of Version 5 Program

5.3 Summary

Now it is time to summarize the essential ideas learned in this chapter. Sometimes you want to run a series of statements more than once with some condition deciding when to stop re-running those statements. When you draw a flowchart that does this, you will see a cycle, or loop, in the flow. Looping is a great way to run some block of code many times. Each time that block, called the loop body, is executed, it is called an iteration of the loop. To determine when to quit the loop, a test can occur anywhere in the loop body, before the loop body, or after the loop body. Using **IF** and **GOTO** it is possible to use any of those three styles of looping. If the test is before the loop body, it is a pre-test loop. If the test is after the loop body, it is a post-test loop. Other loops which have a test in the body or multiple tests to exit do not have special names. They are just called loops. Any of these loop structures can be written in ECMA-55 Minimal BASIC using **IF** and **GOTO**. If you have a pre-test loop that uses a counter, then you can use the **FOR** and **NEXT** statements to write many loops more easily. Note that some common tasks, such as getting user input and verifying it, are naturally post-test loops. It is normal for a non-trivial program to have both some **FOR** loops and some other loops coded using

`IF` and `GOTO`. So why not just use `IF` and `GOTO` every time, since it always works? Using `FOR` loops when possible makes the code easier to read, avoids having to specify line numbers, and makes it clear what part of the program is the loop body. In addition, often using a `FOR` loop your code runs a bit faster since the language translator (compiler or interpreter) will have special optimized code sequences to support `FOR` and `NEXT` that run a little bit faster than using `IF` and `GOTO`.

Exercises

1. Create a flowchart for an algorithm that generates the following sequence: $1, 2, 4, 5, 7, 8, 10, \ldots$
2. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

Chapter 6

More Series

Many math problems can be solved with BASIC. Among the simplest of those are generating series of numbers. Famous series that are generated with computers include factorial numbers and Fibonacci numbers. This chapter will teach you how to generate integer sequences, and that will give you some practice using ECMA-55 Minimal BASIC to solve simple math problems.

6.1 Factorials

One famous function is the factorial function. To compute the factorial of a number n you would use equation 6.1.[1]

$$n! = \prod_{i=1}^{n} i = 1{\times}2{\times}3{\times}4{\times} \ldots {\times}n \tag{6.1}$$

So if we wanted to generate a series of factorial values, how could we do it? We know the first few terms of the series would be:

$$1!, 2!, 3!, 4!, 5!, \ldots$$

Which after the factorial values were computed would be:

$$1, 2, 6, 24, 120, \ldots$$

At first this seems like a very hard problem compared to those in the last chapter, but actually solving this is not difficult if you realize the series can be rewritten.

[1] The $\prod_{i=1}^{n}$ symbol means to loop n times, with i=1 the first time, i=2 the second time, etc. until the last time when i=n. Each time through the loop the term formula is computed. In this case the term formula is simply i, and those term values are multiplied together to form a product.

Just rewrite the series like this:

$$1, (2 \times 1!), (3 \times 2!), (4 \times 3!), (5 \times 4!), \ldots$$

Did you notice that each term is just the last term times a constant that happens to be the term number? Once we know that, we can create the flowchart shown in figure 6.1. The BASIC program to implement that algorithm is shown in figure 6.2 on page 39. The runtime output is shown in figure 6.3 on page 39.

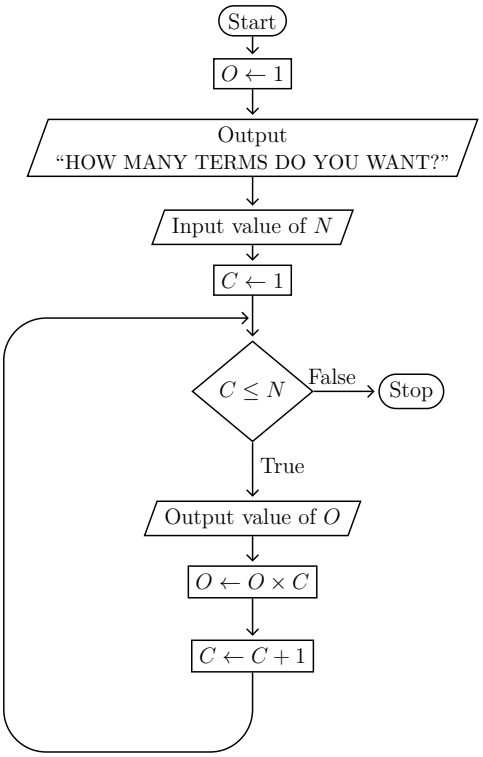


Figure 6.1: Factorial Series Flowchart

```
10 REM COMPUTE FACTORIAL USING ITERATIVE SOLUTION
20 REM O IS THE PRODUCT
30 REM C IS THE CURRENT TERM NUMBER
40 REM N IS THE NUMBER OF TERMS WE WANT
50 LET O=1
60 PRINT "HOW MANY TERMS DO YOU WANT";
70 INPUT N
80 FOR C=1 TO N STEP 1
90 LET O=O*C
100 PRINT O
110 NEXT C
120 END
```

Figure 6.2: Factorial Program Source

```
HOW MANY TERMS DO YOU WANT? 5
 1
 2
 6
 24
 120
```

Figure 6.3: Factorial Program Output

Line Number	Value of C	O	N	Comments
	?	?	?	At the start all values are undefined
10	?	?	?	This is a comment so this line does nothing
20	?	?	?	This is a comment so this line does nothing
30	?	?	?	This is a comment so this line does nothing
40	?	?	?	This is a comment so this line does nothing
50	?	1	?	Program stores value 1 into variable O
60	?	1	?	Program prints prompt message
70	?	1	5	Program reads value of N from keyboard
80	1	1	5	Program stores value 1 into variable C Program checks if value of C which is 1 is greater than value of N which is 5 and it is not so program continues to line 90
90	1	1	5	Program computes value of $O\times$ value of C and stores that into variable O
100	1	1	5	Display value of variable O which is 1
110	2	1	5	Program increments value of loop index C by 1 so the new value is now 2 and then jumps to line 80
80	2	1	5	Program checks if value of C which is 2 is greater than value of N which is 5 and it is not so program continues to line 90
90	2	2	5	Program computes value of $O\times$ value of C and stores that into variable O
100	2	2	5	Display value of variable O which is 2
110	3	2	5	Program increments value of loop index C by 1 so the new value is now 3 and then jumps to line 80
80	3	1	5	Program checks if value of C which is 3 is greater than value of N which is 5 and it is not so program continues to line 90
90	3	6	5	Program computes value of $O\times$ value of C and stores that into variable O
100	3	6	5	Display value of variable O which is 6

Continued on next page

Line Number	Value of C	O	N	Comments
Continued from previous page				
110	4	6	5	Program increments value of loop index C by 1 so the new value is now 4 and then jumps to line 80
80	4	6	5	Program checks if value of C which is 4 is greater than value of N which is 5 and it is not so program continues to line 90
90	4	24	5	Program computes value of $O\times$ value of C and stores that into variable O
100	4	24	5	Display value of variable O which is 24
110	5	24	5	Program increments value of loop index C by 1 so the new value is now 5 and then jumps to line 80
80	5	24	5	Program checks if value of C which is 5 is greater than value of N which is 5 and it is not so program continues to line 90
90	5	120	5	Program computes value of $O\times$ value of C and stores that into variable O
100	5	120	5	Display value of variable O which is 120
110	6	120	5	Program increments value of loop index C by 1 so the new value is now 6 and then jumps to line 80
80	6	120	5	Program checks if value of C which is 6 is greater than value of N which is 5 and it is so program exits the loop and continues on line 120
120	6	120	5	This is an **`END`** statement so the program execution ends

Table 6.1: Factorial Runtime Trace With Input 5

Have you noticed that the series output of the example programs so far is not as nice as it could be? Instead of having multiple comma-delimited terms output on a line, the program instead displays each term on a separate line.

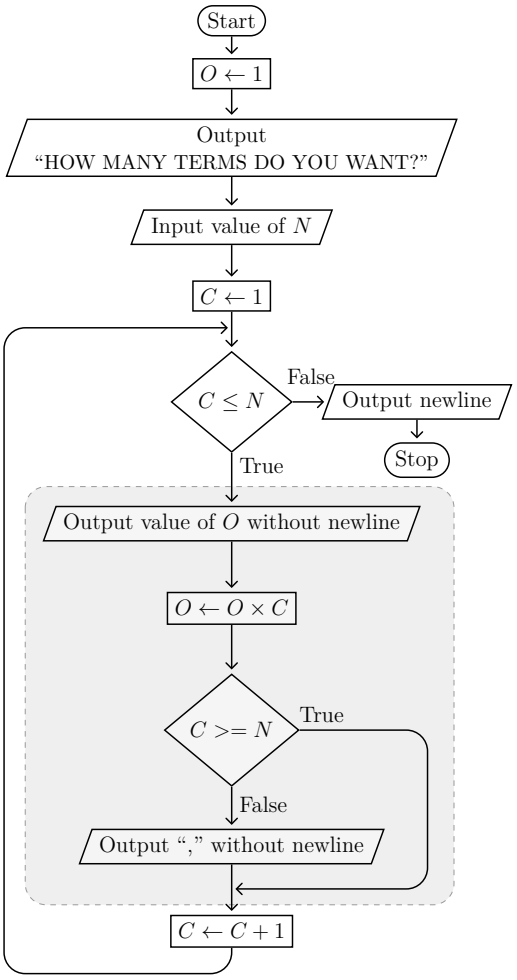


Figure 6.4: Nicer Factorial Series Flowchart

It is possible to do a nicer job of output formatting, but it requires using more logic. ECMA-55 Minimal BASIC will always output a space after a numeric value. For positive numbers there is also a space before a numeric value. This means that the best we can hope for with $N = 5$ is shown in figure 6.5 on page 43.

So what can we do to actually achieve that output? We need to know that the **PRINT** statement always will end with a newline *unless* the last character on the line is a semicolon[2], in which case the output is added to a pending buffer. Also, a **PRINT** with no arguments is valid and will force any partial output in the pending buffer to be output followed by a newline. If the pending buffer is empty, then a blank line will be output.

Armed with that knowledge, we can change the algorithm logic as shown in figure 6.4. The shaded area is called the *body of the loop*. The program created with that flowchart is shown in figure 6.7 on page 43. Note that the body of the loop is implemented with the code between the **FOR** on line 80 and the **NEXT** on line 130. The second diamond is implemented with the **IF** statement on line 110. The **IF** on line 110 must branch to line 130 and not line 80 because the programs need to be allowed to execute the **NEXT** so that the variable C gets incremented correctly.

The output of the program shown in figure 6.7 does give the nicer output if the number of terms is small. If the number of terms is large then the automatic

[2]A comma also has this characteristic, but aligns the output on columns. This is explained in chapter 9.

wrapping algorithm used by the **PRINT** statement can result in ugly output as shown in figure 6.6.[3] Notice the comma at the start of the fourth line, which is truly hideous. To fix this you need to use only 5 columns on any one line to ensure that any size number can always fit. See chapter 9 for a detailed example of how to generate nice-looking five column output.

```
HOW MANY TERMS DO YOU WANT? 5
 1 , 2 , 6 , 24 , 120
```

Figure 6.5: Best Possible Factorial Output

```
HOW MANY TERMS DO YOU WANT? 27
 1 , 2 , 6 , 24 , 120 , 720 , 5040 , 40320 , 362880 , 3.6288E+6 , 3.99168E+7 ,
 4.79002E+8 , 6.22702E+9 , 8.71783E+10 , 1.30767E+12 , 2.09228E+13 ,
 3.55687E+14 , 6.40237E+15 , 1.21645E+17 , 2.4329E+18 , 5.10909E+19 , 1.124E+21
, 2.5852E+22 , 6.20448E+23 , 1.55112E+25 , 4.03291E+26 , 1.08889E+28
```

Figure 6.6: Ugly Factorial Output

```
10 REM COMPUTE FACTORIAL USING ITERATIVE SOLUTION
20 REM O IS THE PRODUCT
30 REM C IS THE CURRENT TERM NUMBER
40 REM N IS THE NUMBER OF TERMS WE WANT
50 LET O=1
60 PRINT "HOW MANY TERMS DO YOU WANT";
70 INPUT N
80 FOR C=1 TO N STEP 1
90 LET O=O*C
100 PRINT O;
110 IF C>=N THEN 130
120 PRINT ",";
130 NEXT C
140 PRINT
150 END
```

Figure 6.7: Nicer Factorial Program Source

6.2 Fibonacci Numbers

One famous series that takes a little more effort to produce is called the Fibonacci sequence. Each term is called a Fibonacci number. This sequence is:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$$

[3]The actual wrapping output may depend on your specific ECMA-55 Minimal BASIC implementation, but if enough terms are output eventually the ugly automatic wrapping will occur.

For each number starting with the 2, you can compute the number with the formula:

$$F_i = F_{i-1} + F_{i-2} \tag{6.2}$$

This means that each term of the series depends on the previous two terms of the series. How can this series be generated with ECMA-55 Minimal BASIC? Actually, the trick is to realize that the first two terms are 1 and the third term only needs two previous terms, so we can generate the third term. Now that we have the third term, we can generate the fourth term. For any term after the first two, all we need is the previous two terms in the series. As long as we start with the first two terms which we know are both 1, we can, in theory, generate the n^{th} term for any n, although machine precision will limit this when n grows large enough. In fact, once you consider this carefully you will realize that for any term F_k, we only need the previous two terms F_{k-1} and F_{k-2} to compute it. So with three variables we can compute the terms of the series using equation 6.2. A flowchart to compute the first ten terms of the Fibonacci Sequence is shown in figure 6.8 on page 45, and a corresponding program is shown in figure 6.9 on page 45. The output of the program is shown in figure 6.10 on page 46 and it is easy to verify that the output is correct. The program is hard-coded to output ten terms, but you can easily modify it to output N terms using the same method that was used in the factorial program earlier in this chapter.

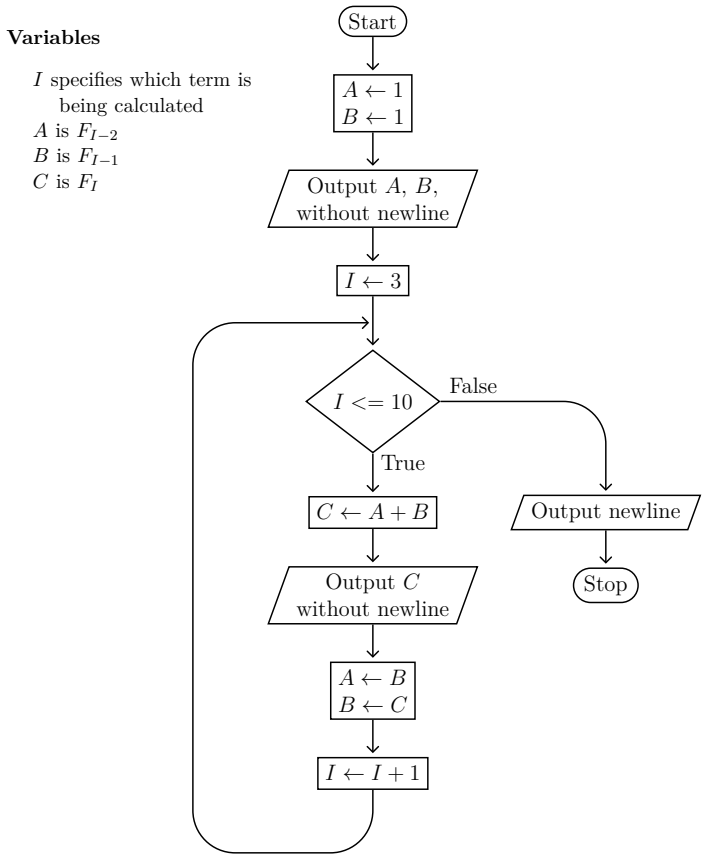


Figure 6.8: Fibonacci Sequence Flowchart

```
10 REM GENERATE FIRST 10 TERMS OF FIBONACCI SEQUENCE
20 REM A IS TERM I-2
30 REM B IS TERM I-1
40 REM C IS TERM I
50 LET A=1
60 LET B=1
70 PRINT A, B,
80 FOR I=3 TO 10
90 LET C=A+B
100 PRINT C,
110 LET A=B
120 LET B=C
130 NEXT I
140 PRINT
150 END
```

Figure 6.9: Fibonacci Sequence Program Source

```
1       1       2       3       5
8       13      21      34      55
```

Figure 6.10: Fibonacci Sequence Program Output

6.3 Taylor Series

Many trigonometric functions can be approximated with a Taylor Series. You can look up the formulas in math reference books or on the Internet. This section will show how to compute the cosine of an angle θ specified in radians. The Taylor series to approximate $cosine(\theta)$, where θ is in radians, is given by:

$$cosine(x) = \sum_{n-0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots \qquad \text{for all x}$$

Note well that each term after the first actually can be computed using the previous term. Consider the third term's numerator. It is simply the previous term's numerator multiplied by x^2. The third term's denominator is a bit trickier, but it is still $2!\times3\times4$, which is really $2!\times2n\times(2n-1)$ with $n = 2$. The last trick is to realize that the need to add or subtract a term alternates, and terms $0, 2, 4, \ldots$ are added, and terms $1, 3, 5, \ldots$ are subtracted. This is tricky, so it will help to show how to generate term_2 from term_1 step-by-step. Remember, the first term is term_0 and is always 1.

$$\begin{aligned} a &= x^2 \\ n_1 &= a \\ d_1 &= 2 \\ t_1 &= \frac{n_1}{d_1} \end{aligned}$$

$$\begin{aligned} n_2 &= n_1 \times a \\ d_2 &= d_1 \times 2n \times (2n-1) \\ t_2 &= \frac{n_2}{d_2} \end{aligned}$$

Now that we see the way to generate any term after term_1 from the previous term, it is clear a solution similar to that used by the Fibonacci sequence is needed, but with a few more variables, and this time we only need to keep information about one previous term instead of two.

The ECMA-55 Minimal BASIC implementation is shown in figure 6.11 on page 48. It works for angles between 0 and $\frac{\pi}{8}$, but larger angles do not work because of precision issues with the floating point math that are beyond the scope of this text.

```
10 REM COMPUTE COS(X) WITH TAYLOR SERIES FOR ANGLE A IN RADIANS
20 PRINT "WHAT IS THE ANGLE IN RADIANS";
30 INPUT A
40 PRINT "HOW MANY TERMS SHOULD I USE";
50 INPUT N
60 LET A2=A*A
70 LET N0=A2
80 LET D0=2
90 LET T=1-N0/D0
100 FOR I=2 TO N
110 LET N1=N0*A2
120 LET J1=2*I
130 LET D1=D0*J1*(J1-1)
140 LET T2=N1/D1
150 IF INT(I/2)<>I/2 THEN 170
160 LET T2=-T2
170 LET T=T+T2
180 NEXT I
190 PRINT COS(A),T
200 END
```

Figure 6.11: Taylor Series Cosine Source

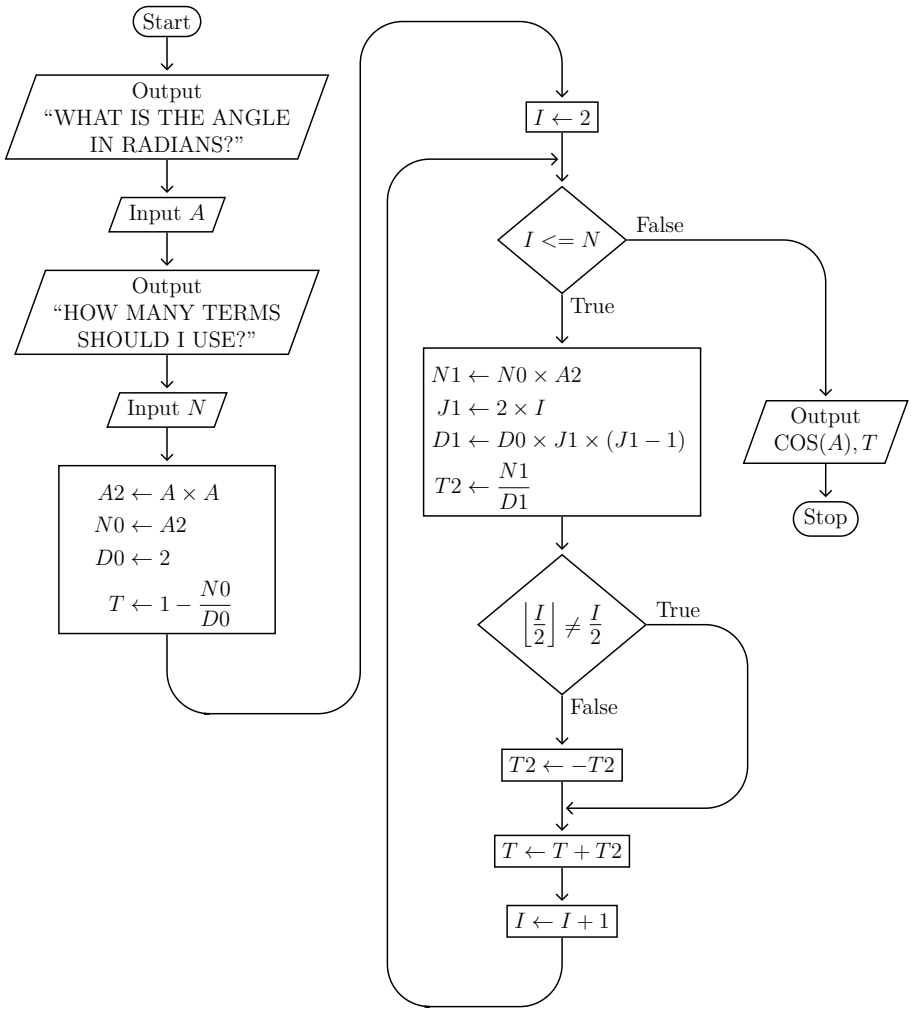


Figure 6.12: Taylor Series Cosine Flowchart

Exercises

1. Create a flowchart for an algorithm that generates the following sequence: $10, -20, 30, -40, 50, -60, \ldots$

2. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

3. The Fibonacci sequence can be extended to the negative numbers if you start with terms 0 and 1 instead of 1 and 1, and you use the formula

$$F_{i-2} = F_i - F_{i-1}$$

   Create a flowchart for the negafibonacci numbers for terms F_{-1} through F_{-10}.

4. Write and test the program for the flowchart in the previous question.

5. The Taylor series to approximate $sine(\theta)$, where θ is in radians, is given by:

$$sine(x) = \sum_{n-0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots \qquad \text{for all x}$$

   Create a flowchart to use this method and then write the program to calculate the sine function using the Taylor series and compare the results it produces with the results of the built-in `SIN()` function for the same angle. What value of n works best for the angle $\frac{\pi}{3}$?

Chapter 7

Random Numbers

One simple game some young people play when learning about mathematics is called high-low. In this two-person game, player one thinks of a random integer value within some range. Player two then tries to guess the number. If player two's guess is lower than player one's value, then player one will say "Too low!" If player two's guess is higher than player one's value, then player one will say "Too high!" When player two's guess is correct, player one will say "Correct!" We will write a program that plays this game as player one, with the user being player two. The first thing to do is create the algorithm. The logic you need is shown in figure 7.1.

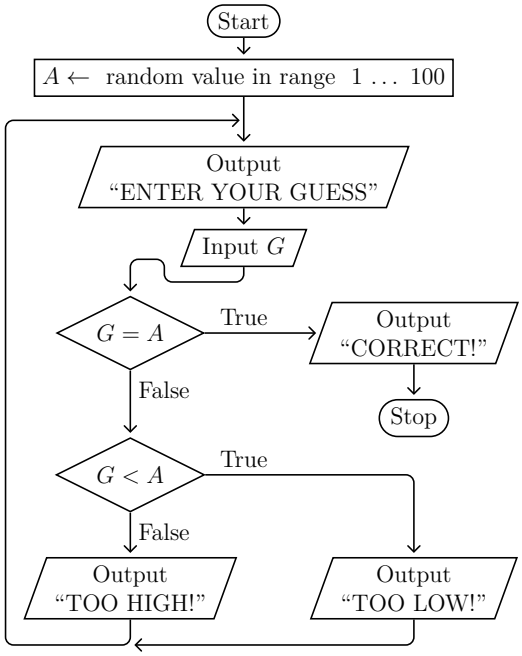


Figure 7.1: High-low flowchart

You already know how to write code for input and output. You have learned about conditional branches, unconditional branches, and assignment. But one thing that has not been explained yet is how to generate a random number. BASIC has a special built-in function for this called `RND`. The `RND` function takes no arguments

and will return a value in the range $0 \leq X < 1$ that is a pseudo-random number. A pseudo-random number is not truly random, only an approximation of a random number. However, it is a good enough approximation for most games and we will use it. In our game, we want to limit the values that player one chooses to be between 1 and 100. To achieve this the program will use a formula like this:

$$A = 1 + \text{INT}(\text{RND} \times 100)$$

This works since if **RND** returns something like 0.99999 then when we multiply by 100 we get 99.999 and the **INT()** function will then return 99 and we add 1 and get 100, the maximum value we wanted. The **INT(X)** function takes the argument **X** and returns the floor of that value.[1] It is a *built-in* function for BASIC. At the other end of the range, if RND returns 0, we add 1 and get 1, the minimum value we wanted. With that knowledge, it is possible to write the program shown in figure 7.2.

```
10 LET A=1+INT(RND*100)
20 PRINT "ENTER YOUR GUESS";
30 INPUT G
40 IF G=A THEN 100
50 IF G>A THEN 80
60 PRINT "TOO LOW!"
70 GOTO 20
80 PRINT "TOO HIGH!"
90 GOTO 20
100 PRINT "CORRECT!"
110 END
```

Figure 7.2: High-Low Program

There remains only one problem. Every time you run this game, it picks the same random value. That's not very fun now is it? This is a feature of BASIC that people find strange the first time they see it. By default, BASIC will always generate the same series of random numbers. This is so that you can test your program and know you will get exactly the same results every time. When you want truly random behavior, you need to add a special statement at the beginning of the program called **RANDOMIZE**. We also would like to have the program remember how many guesses the human player needed to guess the number and to tell the human player that number once they have finally guessed correctly. A flowchart that can do that is shown in figure 7.3 on page 53, and the corresponding program is shown in figure 7.4 on page 54. Please notice that this program is better than the first version since it includes comments describing the program and the variables

[1] In math texts, the function floor(x) is written $\lfloor x \rfloor$ and is defined as the largest integer not greater than x, so floor(.9) is 0, floor(-.9) is -1, floor(1.1) is 1, floor(-1.1) is -2, etc. Applying the floor function to an integer returns that same integer value, so floor(3) is 3 and floor(-3) is -3.

that are used. Also after careful inspection you will see that while the **RANDOMIZE** statement is included in the program source code, it is not shown in the flowchart. This is because that is an implementation detail of BASIC. In a flowchart, you can assume that if it says *random value* that the intent is to get a value as random as possible. To turn that feature on in ECMA-55 Minimal BASIC, the **RANDOMIZE** is needed.

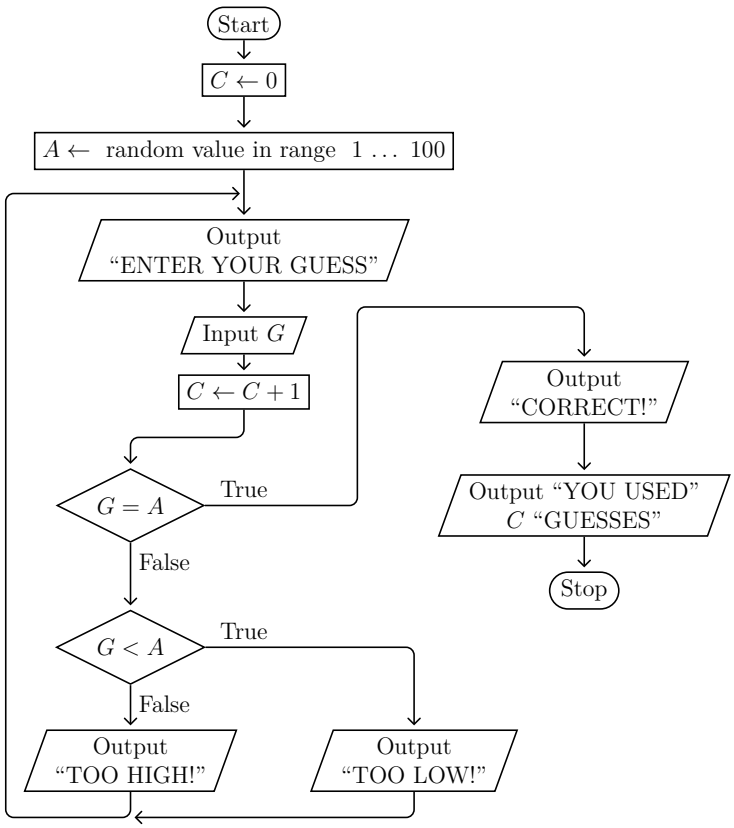


Figure 7.3: Improved High-low flowchart

You can see a sample run of the program in figure 7.5. Can you see a pattern to the guesses? The number of guesses *required* is never more than seven. We will learn why in chapter 15.

```
10 REM HIGH-LOW GUESSING GAME
20 REM A IS THE ANSWER THE COMPUTER PLAYER HAS CHOSEN
30 REM G IS THE GUESS THE HUMAN PLAYER HAS CHOSEN
40 REM C IS THE NUMBER OF GUESSES THE HUMAN PLAYER HAS MADE
50 RANDOMIZE
60 LET C=0
70 LET A=1+INT(RND*100)
80 PRINT "I HAVE CHOSEN AN INTEGER VALUE BETWEEN 1 AND 100.  TRY AND"
90 PRINT "GUESS WHAT NUMBER I HAVE CHOSEN."
100 PRINT "ENTER YOUR GUESS";
110 INPUT G
120 LET C=C+1
130 IF G=A THEN 190
140 IF G>A THEN 170
150 PRINT "TOO LOW!"
160 GOTO 100
170 PRINT "TOO HIGH!"
180 GOTO 100
190 PRINT "CORRECT!"
200 PRINT "YOU USED";C;"GUESSES."
210 END
```

Figure 7.4: Improved High-Low Program

```
I HAVE CHOSEN AN INTEGER VALUE BETWEEN 1 AND 100.  TRY AND
GUESS WHAT NUMBER I HAVE CHOSEN.
ENTER YOUR GUESS? 50
TOO HIGH!
ENTER YOUR GUESS? 25
TOO HIGH!
ENTER YOUR GUESS? 12
TOO LOW!
ENTER YOUR GUESS? 19
CORRECT!
YOU USED 4 GUESSES.
```

Figure 7.5: Improved High-Low Program Output

Exercises

1. Create a flowchart for the program shown in figure 7.4 on page 54 by starting with figure 7.1 on page 51 and adding the new logic for counting the number of guess the human player has made and reporting that number at the end of the game.

2. Change the program so that $100 \leq A < 200$.

Chapter 8

Multi-way Branching

In this chapter you will learn to how implement multi-way branching. A *multi-way branch* is a branch that has more than two possible destinations. When drawn with a flowchart, the diamond symbol is used, but unlike a normal conditional branch which has only two outgoing edges, a multi-way branch has more than two outgoing edges. An ECMA-55 Minimal BASIC **`IF`** statement implements a normal conditional branch, so it has only two destinations: the jump target specified after the **`THEN`** keyword when the condition is true, and the next line in the program when the condition is false. A multi-way branch must support more than just two destinations.

An example of a simple flowchart fragment for a five-way branch is shown in figure 8.1. When the expression within the diamond is evaluated, instead of just returning true or false, it will return one of five values in the set $\{1, 2, 3, 4, 5\}$. These outputs are labeled with Case 1, Case 2, ... Case 5 in the flowchart.

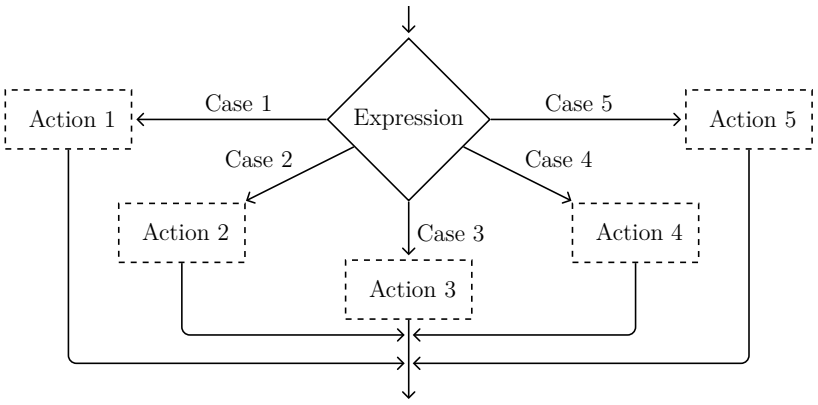


Figure 8.1: Five-way branch flowchart

This flowchart uses process boxes with dashes instead of solid lines for the border. This nested code symbol represents potentially nested code, but could be a single output or assignement statement, or something more complex like a subroutine call or a sequence of statements.

8.1 Implementation of multi-way branch using `IF`

A multi-way branch can be implemented using a series of conditional branch statements, as shown in the flowchart in figure 8.2.

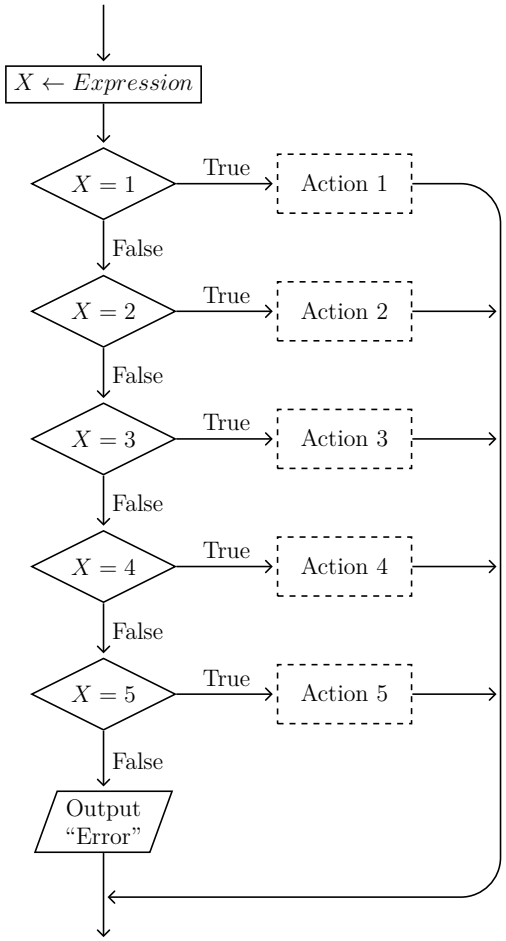


Figure 8.2: Five one-way branches flowchart

An ECMA-55 Minimal BASIC program which implements the flowchart (just printing ACTION # for each action) is shown in figure 8.3 on page 59. It requires, in the worst case, evaluating all five conditions. The worst case occurs when a user enters a value that is invalid. An alternate solution with equivalent semantics using a style that some people prefer more is shown in figure 8.4 on page 59. The alternate solution groups each action with its corresponding condition, while the first solution groups the conditions and actions into two different sections. Which style to use is a matter of personal preference, since neither style is clearly superior.

```
10 PRINT "ENTER A NUMBER BETWEEN 1 AND 5 (INCLUSIVE)";
20 INPUT X
30 IF X=1 THEN 100
40 IF X=2 THEN 120
50 IF X=3 THEN 140
60 IF X=4 THEN 160
70 IF X=5 THEN 180
80 PRINT "INVALID INPUT!"
90 GOTO 10
100 PRINT "ACTION 1"
110 GOTO 190
120 PRINT "ACTION 2"
130 GOTO 190
140 PRINT "ACTION 3"
150 GOTO 190
160 PRINT "ACTION 4"
170 GOTO 190
180 PRINT "ACTION 5"
190 END
```

Figure 8.3: Five one-way IF statements

```
10 PRINT "ENTER A NUMBER BETWEEN 1 AND 5 (INCLUSIVE)";
20 INPUT X
30 IF X<>1 THEN 60
40 PRINT "ACTION 1"
50 GOTO 200
60 IF X<>2 THEN 90
70 PRINT "ACTION 2"
80 GOTO 200
90 IF X<>3 THEN 120
100 PRINT "ACTION 3"
110 GOTO 200
120 IF X<>4 THEN 150
130 PRINT "ACTION 4"
140 GOTO 200
150 IF X<>5 THEN 180
160 PRINT "ACTION 5"
170 GOTO 200
180 PRINT "INVALID INPUT!"
190 GOTO 10
200 END
```

Figure 8.4: Five one-way IF statements (alternate solution)

8.2 Implementation of multi-way branch using ON...GOTO

While the technique just shown works, it is not very efficient if you have a large number of outgoing branches. For this reason, there is a more direct way to implement multi-way branches in the ECMA-55 Minimal BASIC language. You can use the **ON...GOTO** statement. The syntax of the ECMA-55 Minimal BASIC statement is show in figure 8.5.

```
ON numeric-expression GOTO line-number, ... ,line-number
```

Figure 8.5: **ON** ... **GOTO** statement

This statement will evaluate the expression and if the value is 1, it will jump to the first line number in the list. If the value is 2, it will jump to the second line number in the list. If the value is less than 1 or greater than the number of line numbers in the list, the program will exit with an error message. This requires that you ensure the expression can never have a value that is less than 1 or greater than the number of elements in the comma-delimited list of line numbers specified after the **GOTO** keyword.

The ECMA-55 Minimal BASIC program in figure 8.6 implements the same semantics as the program shown in figure 8.3 on page 59, but takes advantage of the true multi-way branch support provided by the **ON...GOTO** statement.

```
10 PRINT "ENTER A NUMBER BETWEEN 1 AND 5 (INCLUSIVE)";
20 INPUT X
30 LET X=INT(X)
40 IF X<1 THEN 170
50 IF X>5 THEN 170
60 ON X GOTO 70,90,110,130,150
70 PRINT "ACTION 1"
80 GOTO 190
90 PRINT "ACTION 2"
100 GOTO 190
110 PRINT "ACTION 3"
120 GOTO 190
130 PRINT "ACTION 4"
140 GOTO 190
150 PRINT "ACTION 5"
160 GOTO 190
170 PRINT "INVALID INPUT!"
180 GOTO 10
190 END
```

Figure 8.6: One five-way **ON...GOTO** statement

Note that it still requires three comparisons in the best case since we must check that the value is not too low on line 40 and not too high on line 50. But in the

worst case, we still use just three comparisons, no matter how many outgoing edges the multi-way branch has. If you have more than 3 possibilities, it is better to use the `ON...GOTO` statement than a series of `IF` statements.

There is one weakness, however, that must be mentioned. The expression evaluated in the `ON...GOTO` statement must be numeric. ECMA-55 Minimal BASIC's multi-way branch statement does not work for string expressions. If you ever find you need to implement a multi-way branch and the condition has a string type, you *must* use multiple `IF` statements in that case.

Exercises

1. Draw a flowchart with a seven-way branch where each branch is for each day of the week as a numeric value, with 1=Monday, 2=Tuesday, etc.
2. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.
3. Draw a flowchart with a seven-way branch where each branch is for each day of the week as a string value such as “MONDAY”, “TUESDAY”, etc.
4. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

Chapter 9

Multicolumn Output

In this chapter you will learn to how generate nice five column output. The ECMA-55 Minimal BASIC language supports five output columns with the comma delimiter of the **PRINT** statement. To make good looking tables with five columns, this feature should be used together with logic to ensure no extra blank lines are printed. The trick is that for the first four columns, the number is printed and the **PRINT** statement uses a trailing comma, but for the fifth column no trailing comma is used, forcing a newline.

9.1 First Draft

To accomplish this, we need more condition tests. This results in a more complex flowchart as shown in figure 9.1. Now there are three diamonds. One of those is taken care of with the **FOR** statement, but the other two will require **IF** statements.

An implementation of the flowchart in figure 9.1 on page 64 is shown in figure 9.2 on page 65. You can see the test for whether we are printing the fifth column or not on lines 40 and 120. Line 70 has the correct **PRINT** with the trailing comma to use for the first four columns and line 50 has the **PRINT** without the trailing comma for the fifth column. The bare **PRINT** statement on line 130 will only be used if the test on line 120 shows that the last value printed was *not* in the fifth column. This condition is reversed on line 120 since we want to skip over the bare **PRINT** if the last column printed was the fifth column, whereas the test on line 40 wants to fall through to the bare print if we *are* on the fifth column. The condition **A/B = INT(A/B)** will be true if A is an integer multiple of B for any non-zero, positive value of B.[1] Remember, the **INT()** function is the mathematical $floor(x)$ function, as explained in a footnote[2] in chapter 7.

[1] Some people do this with a modulus by testing if A *modulo* $B = 0$, but ECMA-55 Minimal BASIC does not support a modulo or an integer remainder operator.

[2] See the footnote about floor on page 52.

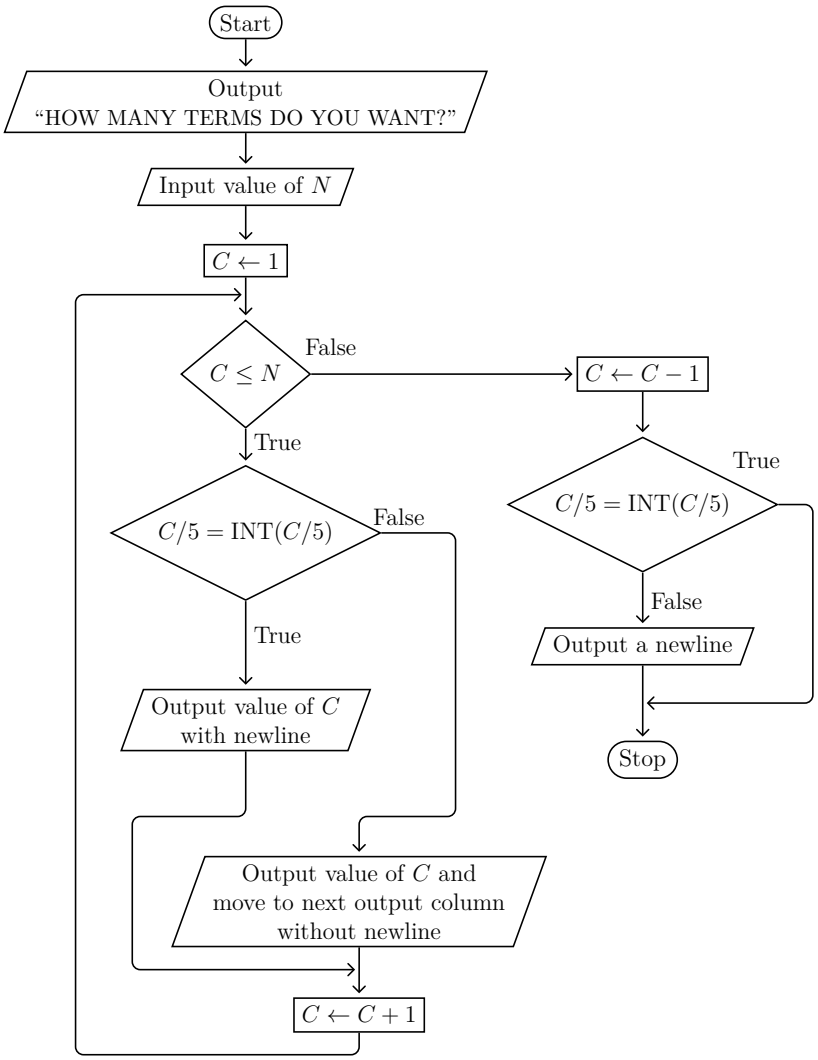


Figure 9.1: Five column program flowchart

When testing the program in figure 9.2 on page 65, we must test both the case where N is an exact multiple of five and when it is not. The case where N *is not* an exact multiple of five is shown in figure 9.3 on page 65, and the case where N *is* an exact multiple of five is shown in figure 9.4 on page 65.

```
10 PRINT "HOW MANY TERMS DO YOU WANT TO PRINT";
20 INPUT N
30 FOR C=1 TO N STEP 1
40 IF C/5<>INT(C/5) THEN 70
50 PRINT C
60 GOTO 80
70 PRINT C,
80 NEXT C
90 REM THE FINAL NEXT ADDS ONE TO C BREAKING THE COUNT
100 REM SO SUBTRACT ONE NOW TO COMPENSATE FOR THAT
110 LET C=C-1
120 IF C/5=INT(C/5) THEN 140
130 PRINT
140 END
```

Figure 9.2: Program for five column output

```
HOW MANY TERMS DO YOU WANT TO PRINT? 17
 1             2             3             4             5
 6             7             8             9             10
 11            12            13            14            15
 16            17
```

Figure 9.3: Five column output when not a multiple of 5

```
HOW MANY TERMS DO YOU WANT TO PRINT? 15
 1             2             3             4             5
 6             7             8             9             10
 11            12            13            14            15
```

Figure 9.4: Five column output for a multiple of 5

9.2 Second Draft

Now that you have seen a way to print using the five columns available in ECMA-55 Minimal BASIC, it is time to investigate how to convert the specific solution that just prints the sequence of counting numbers into a more general solution that can display any sequence you can calculate. Fortunately, the changes are very small. Instead of directly printing the loop index C, and new variable T is used to hold the term value. So instead of printing C the program will be printing T that is calculated at the top of the loop body *before* the logic about printing in columns. As an example, consider a program to compute this series:

$$2, 5, 10, 17, 26, \ldots$$

The terms of the series would be computed using formula 9.1, where the term is a function of the loop index C and C ranges from 1 to some upper bound N where N is the number of terms in the series that you want to print.

$$T = C^2 + 1 \tag{9.1}$$

The updated flowchart that can implement this idea is shown in figure 9.5 on page 67, and the corresponding BASIC program is shown in figure 9.6 on page 68. This solution will work well for any term that can be calculated with a simple formula. The term T is computed with a **`LET`** statement on line 50, and the **`PRINT`** statements on lines 70 and 90 have been updated to display the term T instead of the loop index C. Now any time you want to change the formula you can update just one line, line 50, and nothing else in the program will need to change.

This new program still does not allow us to start at any arbitrary term but instead only allows starting with C equal to one. A program where the limits are constants like 1 instead of variables limits how we can use the program. When a program has a constant literal value in the logic instead of a using variable, computer programmers will say that the program is *hard-coded*. In BASIC, the **`FOR`** loop step is also hard-coded to be 1 by default. However, you can use the **`STEP`** keyword, together with a variable or arithmetic expression, to override that.

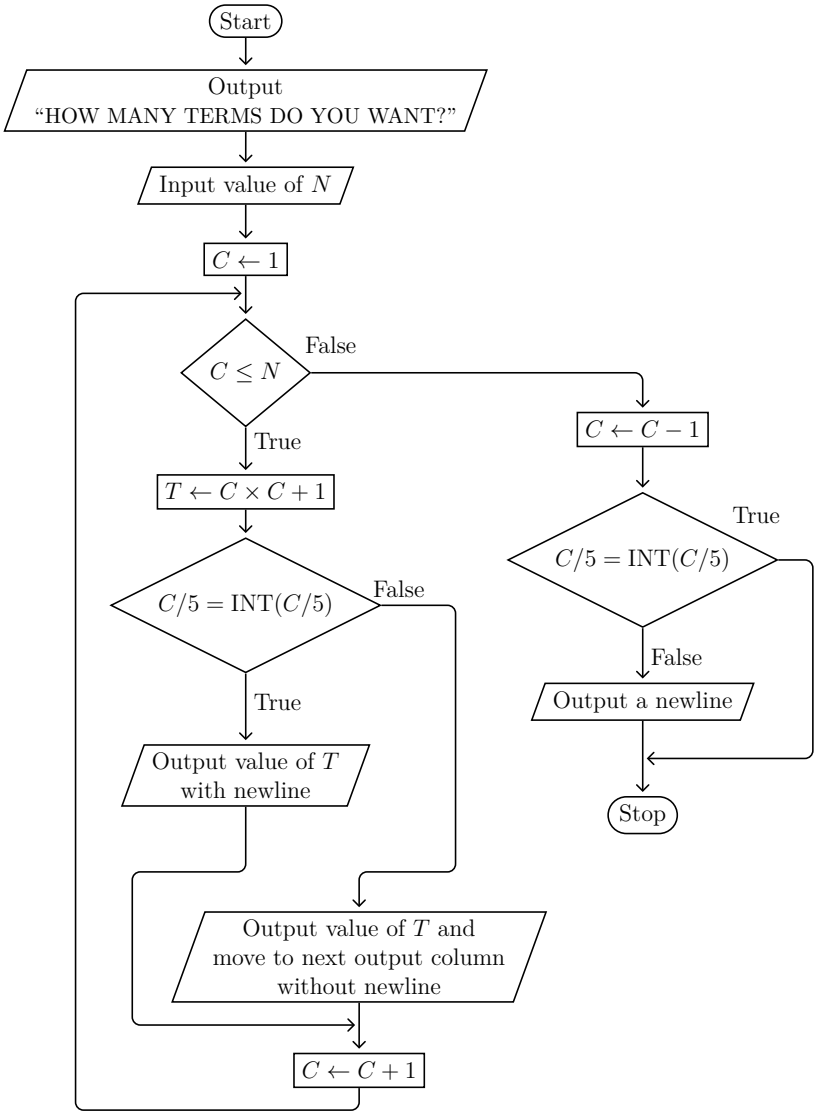


Figure 9.5: Improved five column program flowchart

```
10 PRINT "HOW MANY TERMS DO YOU WANT TO PRINT";
20 INPUT N
30 FOR C=1 TO N STEP 1
40 REM COMPUTE TERM T
50 LET T=C*C+1
60 IF C/5<>INT(C/5) THEN 90
70 PRINT T
80 GOTO 100
90 PRINT T,
100 NEXT C
110 REM THE FINAL NEXT ADDS ONE TO C BREAKING THE COUNT
120 REM SO SUBTRACT ONE NOW TO COMPENSATE FOR THAT
130 LET C=C-1
140 IF C/5=INT(C/5) THEN 160
150 PRINT
160 END
```

Figure 9.6: Improved program for five column output

9.3 Third Draft

We learned in a previous chapter that we can do better. Let's update the program to use the variable S for the initial value of the loop index C and D for the amount to increment the loop index C after each iteration of the loop. The flowchart will have the same shape, and the program will be very similar too. To make the program easier to use, we will ask for each of the variables the user must specify separately. The program needs the user to specify values for N, S, and D, so there will be three prompts and three reads from the keyboard. This makes the flowchart seem much longer but there are no new diamonds in the flowchart so the new logic is just straight-line code. The improved flowchart is shown in figure 9.7 on page 70, and the improved BASIC program is shown in figure 9.8 on page 71.

Everything should be great, but when we run this program there are **still** some problems. Look at the sample session shown in figure 9.9 on page 71. It is easy to see that there are not 17 terms. Also, the 10 should have been in the third column with the first two columns being blank. It did start at the correct term, and it is skipping every other term as intended by setting the loop increment to 2, so the program is *almost* working. However, it quits before all 17 terms are displayed.

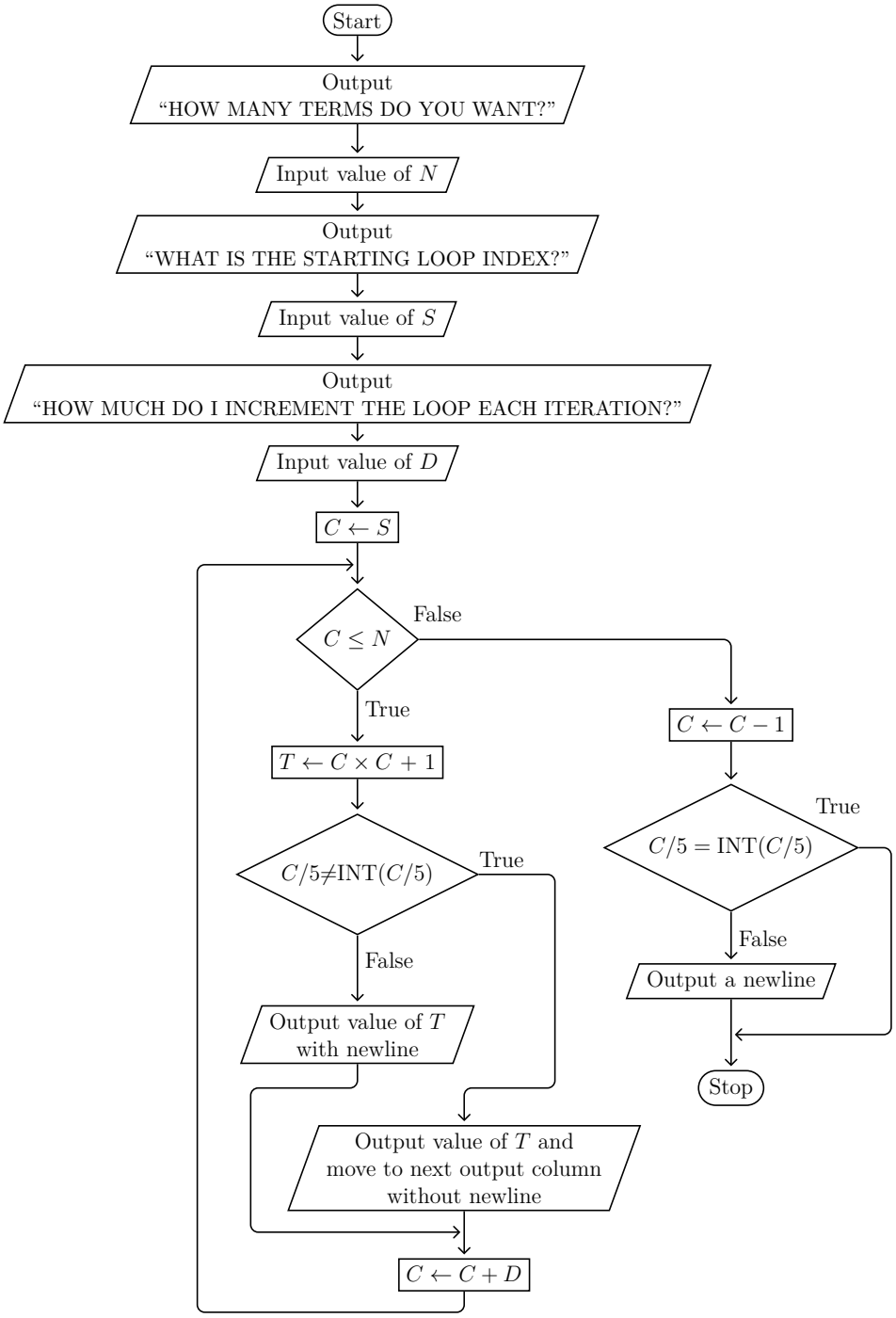


Figure 9.7: Even better five column program flowchart

```
10 PRINT "HOW MANY TERMS DO YOU WANT TO PRINT";
20 INPUT N
21 PRINT "WHAT IS THE STARTING LOOP INDEX";
22 INPUT S
23 PRINT "HOW MUCH DO I INCREMENT THE LOOP EACH ITERATION";
24 INPUT D
30 FOR C=S TO N STEP D
40 REM COMPUTE TERM T
50 LET T=C*C+1
60 IF C/5<>INT(C/5) THEN 90
70 PRINT T
80 GOTO 100
90 PRINT T,
100 NEXT C
110 REM THE FINAL NEXT ADDS ONE TO C BREAKING THE COUNT
120 REM SO SUBTRACT ONE NOW TO COMPENSATE FOR THAT
130 LET C=C-1
140 IF C/5=INT(C/5) THEN 160
150 PRINT
160 END
```

Figure 9.8: Even better program for five column output

```
HOW MANY TERMS DO YOU WANT TO PRINT? 17
WHAT IS THE STARTING LOOP INDEX? 3
HOW MUCH DO I INCREMENT THE LOOP EACH ITERATION? 2
 10             26
 50             82             122            170            226
 290
```

Figure 9.9: Five column failure

9.4 Final Program

The first bug to fix is the failure to start showing the first term in the correct column. We need to add some more logic before the loop but after the user has entered all of the values for N, S, and D. The new logic needs to output blank columns if we do not start on a term that would go in column 1. Positive terms that can start in column 1 must have equation 9.2 true.

$$(C \ modulo \ 5) + 1 = 1 \tag{9.2}$$

The bug fixes will result in an even larger flowchart. When a flowchart has to span more than one page, circle connectors are used. An arrow into a circle connector means the logic continues on another page. An arrow out of a circle connector means the logic is continuing from a different page. Each connector with arrows going outward must have a unique identifier in the circle, usually a single upper-case letter for small flowcharts. It is invalid if an arrow goes into a connector with a symbol when no corresponding connector with that symbol and arrows going outward exists. Circle connectors can even be used on a single page if there is no way to draw a connecting line that does not intersect with any other connecting line.

It is time to design the new logic that will emit blank columns on the first line when required. Since there are five columns, we know there are five cases to consider as shown in table 9.1.

Value of (INT(C/5))	Choice Number	Number of Columns to skip	BASIC source code
0	1	4	`PRINT ,,,,`
1	2	0	Do nothing
2	3	1	`PRINT ,`
3	4	2	`PRINT ,,`
4	5	3	`PRINT ,,,`

Table 9.1: Column Skip Table

The expression you need is given in equation 9.3.

$$1 + \text{INT}(C) - 5 \times \text{INT}(\text{INT}(C)/5) \tag{9.3}$$

The other bug was that we did not print enough terms. This is because the loop needs to run N times, but if we skip by any value of D except 1 it gets tricky. The

best solution is to have the loop index variable C start at S and increment by 1 for N times. For computing the terms to display, keep in mind that basically we want to display Y for some X where X is actually a function of C. We will use a new variable X to hold the value that is a function of C. Now the column logic can depend on C, and the loop will execute N times, but the Y we display will still be correct.

The updated flowchart that includes those changes is shown in two pieces, one in figure 9.10 on page 74 and the other in figure 9.11 on page 75. The circle with the "A" inside is the connector between the two pieces. The flowchart was too big to draw on one page, so it was split into two pieces and uses a connector.

Now that the flowchart is done, it must be converted into ECMA-55 Minimal BASIC source code. A flowchart for a five-way branch should be implemented with an `ON...GOTO` statement. The trick was to determine what the expression must be, and we already did that by examining the column skip table in figure 9.1 on page 72.

The resulting BASIC program is shown in figure 9.12 on page 76. This BASIC program has logic that is quite tricky so there are plenty of comments included using `REM` statements. Finally this version of the program now produces the correct output as shown in figure 9.13 on page 77.

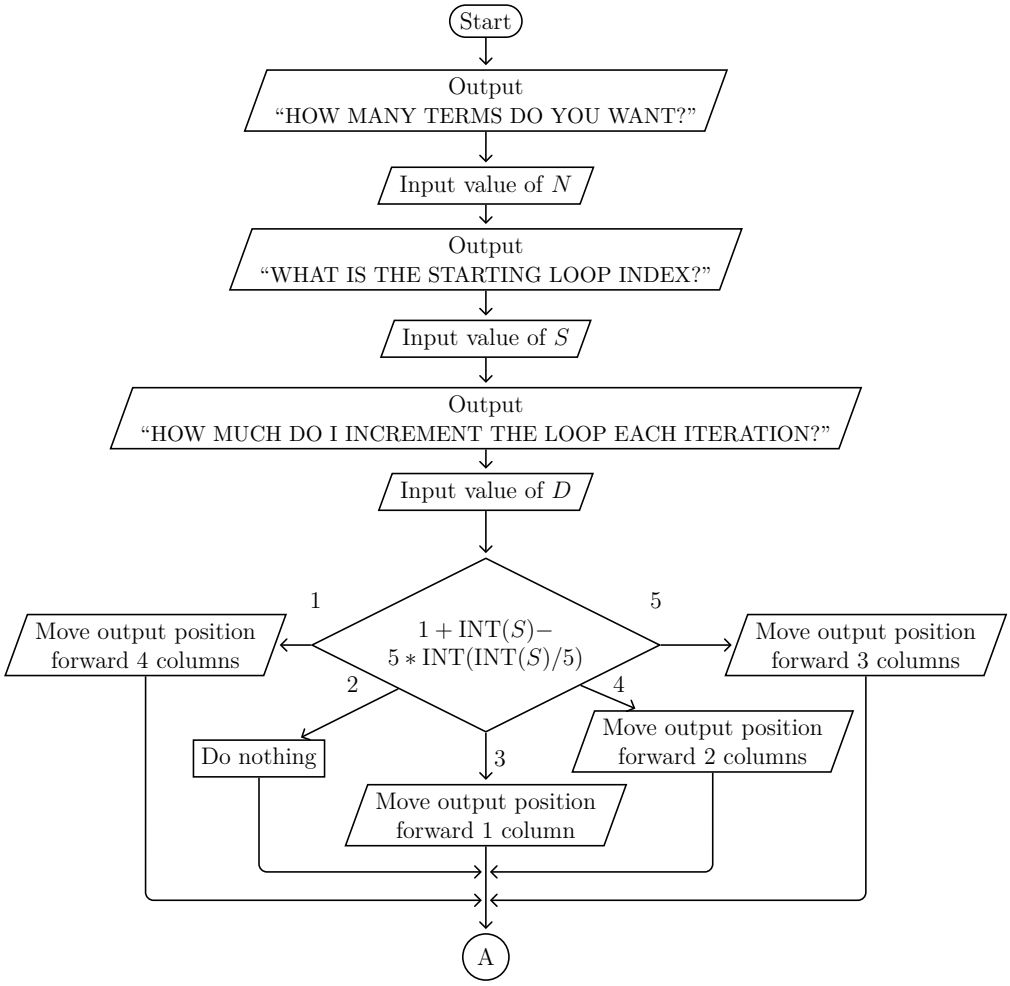


Figure 9.10: Full-featured multicolumn program flowchart (1 of 2)

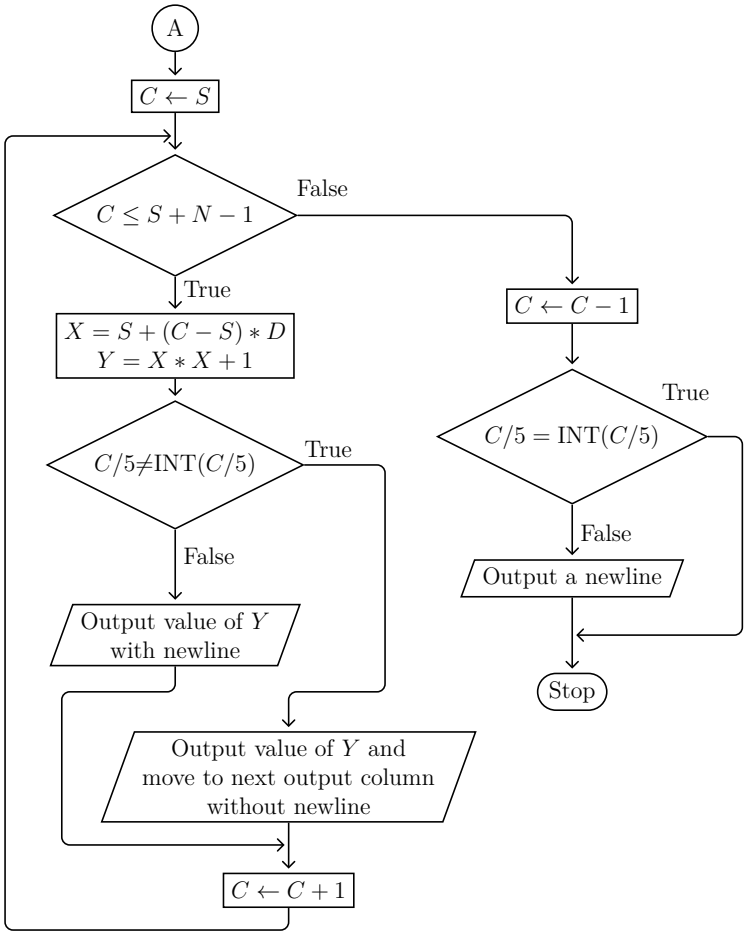


Figure 9.11: Full-featured multicolumn program flowchart (2 of 2)

```
10 PRINT "HOW MANY TERMS DO YOU WANT TO PRINT";
20 INPUT N
30 PRINT "WHAT IS THE STARTING LOOP INDEX";
40 INPUT S
50 PRINT "HOW MUCH DO I INCREMENT THE LOOP EACH ITERATION";
60 INPUT D
70 REM
80 REM VERY COUNTER-INTUITIVE BUT WE NEED TO THINK ABOUT IT.
90 REM INT(S)-5*INT(INT(S)/5) HAS A RANGE 0..4 BUT WE NEED
100 REM A RANGE 1..5.  SO WE ADD ONE TO THE EXPRESSION AND GET
110 REM 1+INT(S)-5*INT(INT(S)/5), AND THAT DOES HAVE THE CORRECT
120 REM RANGE, BUT WE WANT S=1 TO MEAN SKIP 0 COLUMNS, S=2 TO MEAN
130 REM SKIP 1 COLUMN, ETC., SO WE BETTER MAKE A CHART:
140 REM   S  EXPRESSION  ACTION            LINE
150 REM   1      2       DO NOTHING        400
160 REM   2      3       SKIP 1 COLUMN     280
170 REM   3      4       SKIP 2 COLUMNS    310
180 REM   4      5       SKIP 3 COLUMNS    340
190 REM   5      1       SKIP 4 COLUMNS    370
200 REM PUTTING LINE NUMBERS IN EXPRESSION ORDER WILL YIELD
210 REM THE LIST ON LINE 270
220 REM
230 ON 1+INT(S)-5*INT(INT(S)/5) GOTO 330,360,240,270,300
240 REM SKIP 1 COLUMN
250 PRINT ,
260 GOTO 370
270 REM SKIP 2 COLUMNS
280 PRINT ,,
290 GOTO 370
300 REM SKIP 3 COLUMNS
310 PRINT ,,,
320 GOTO 370
330 REM SKIP 4 COLUMNS
340 PRINT ,,,,
350 GOTO 370
360 REM DO NOTHING AND FALL THROUGH
370 REM DONE WITH MULTI-WAY BRANCH
380 REM
390 REM C IS THE COUNTER THAT WILL INCREMENT BY 1 FROM S TO S+N-1
400 REM   FOR A TOTAL OF N ITERATIONS
410 REM X IS THE X FOR OUR EQUATION F(X)=X^2+1
420 REM Y IS F(X) THAT WE COMPUTE AND PRINT
430 REM COLUMN LOGIC USES C, BUT Y COMPUTATION
440 REM USES X WHICH ITSELF IS A FUNCTION OF S,C, AND D
450 FOR C=S TO S+N-1 STEP 1
460 REM COMPUTE CURRENT X=G(C)
470 LET X=S+(C-S)*D
480 REM COMPUTE TERM Y=F(X)=F(G(C))
490 LET Y=X*X+1
500 IF C/5<>INT(C/5) THEN 530
510 PRINT Y
520 GOTO 540
530 PRINT Y,
540 NEXT C
550 REM THE FINAL NEXT ADDS 1 TO C BREAKING THE COUNT
560 REM SO SUBTRACT 1 NOW TO COMPENSATE FOR THAT
570 LET C=C-1
580 IF C/5=INT(C/5) THEN 600
590 PRINT
600 END
```

Figure 9.12: Full-featured multicolumn program

```
HOW MANY TERMS DO YOU WANT TO PRINT? 17
WHAT IS THE STARTING LOOP INDEX? 3
HOW MUCH DO I INCREMENT THE LOOP EACH ITERATION? 2
                              10              26              50
 82             122            170             226             290
 362            442            530             626             730
 842            962            1090            1226
```

Figure 9.13: Full-featured Multicolumn Program Output

Exercises

1. Design a flowchart for a full-featured program that uses just four columns.
2. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

Chapter 10

Arrays

Scalar variables are ideal for holding single values, but often you need to solve problems that involve groups of similar data. A common example is keeping a short list in a computer, such as all of the grades for a quiz, so that you can generate a report or do some statistical analysis.

Most programming languages provide some built-in data structure to help you do this, and ECMA-55 Minimal BASIC is no exception. The BASIC data type that allows programming with groups of data is called an *integer-indexed array*. Since this is the only kind of array supported by ECMA-55 Minimal BASIC, we'll just use the shorter name *array* to refer to integer-indexed arrays in this book. The concept of an array is simple. An array is a contiguous block of memory divided into many pieces of the exactly the same size. Each piece is called an *element*. To access an element you use a number, called a *subscript*, to specify precisely which of the array elements to use. Another common name for a subscript is an *index*. The first element is accessed by using subscript 0. The second element is accessed by using subscript 1, etc. For an array with 9 elements, the subscripts are $0, 1, 2, \ldots, 8$. In mathematics the array concept is sometimes referred to as a *vector*. In figure 10.1 the array name is on the left and is `A`, and the gray boxes represent the memory storage locations for the array. The numbers within those boxes are the values stored in those memory locations. The small numbers across the top are the indices. In this example the third element is 4 and is located at index 2, and the last element is at index 8 and has the value 5. Computer scientists refer to arrays that begin at index 0 as *zero-based arrays*. The C and C++ languages both use zero-based arrays.

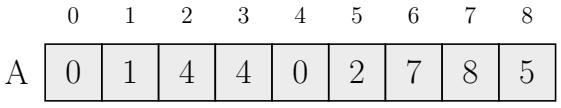


Figure 10.1: Zero-based Array A with 9 elements

In BASIC, arrays have single-letter names from A to Z. It is a fatal error to use the same single-letter variable name for both a scalar variable and an array variable. The subscript expression is specified by appending a parenthesized arithmetic

expression to the array letter. For instance, to access the third element, which is element 2, you would use `A(2)`. Instead of a constant, an expression can be used. For example, `A(I+2)` would calculate the value of the expression `I+2` at runtime, and then use that as the subscript. If `I` had the value 3, then this would get element 5, or the sixth element of the array.

In ECMA-55 Minimal BASIC, by default an array implicitly has 11 elements with indices $0, 1, 2, \ldots 10$. It is possible to use a different size by using a special statement, the `DIM` statement, to declare the size explicitly. However, the `DIM` statement must occur before any use of the array in the program. The `DIM` statement specifies the maximum subscript value permitted, not the number of elements. It is not permitted to explicitly dimension any array more than once in a program. Wise programmers put the `DIM` statements near the beginning of the program.

10.1 Average Value

To better understand how to use arrays in BASIC, some examples are needed. The first example will compute the average value of elements in an array. The formula to compute the average value of all elements in an array A with n elements ranging from $A_0 \ldots A_{n-1}$ is shown in equation 10.1.

$$Average = \frac{\sum_{i=0}^{n-1} A_i}{n} \tag{10.1}$$

Once the problem is considered, it is easy to see that the required algorithm will have three parts. In the first phase it must get the data values from the user to populate the array. In the second phase it will process that data to compute the average. In the third and final phase it will display the average value. A flowchart for one possible solution is shown in figure 10.2 on page 81. The corresponding ECMA-55 Minimal BASIC program is shown in figure 10.3 on page 82. Note that the `DIM` statement specifies the value 7 as its argument. The argument is the number of the maximum subscript possible, not the number of elements directly. Since elements start at 0, `DIM(7)` means that there are 8 elements. A sample run of the program is shown in figure 10.4 on page 82.

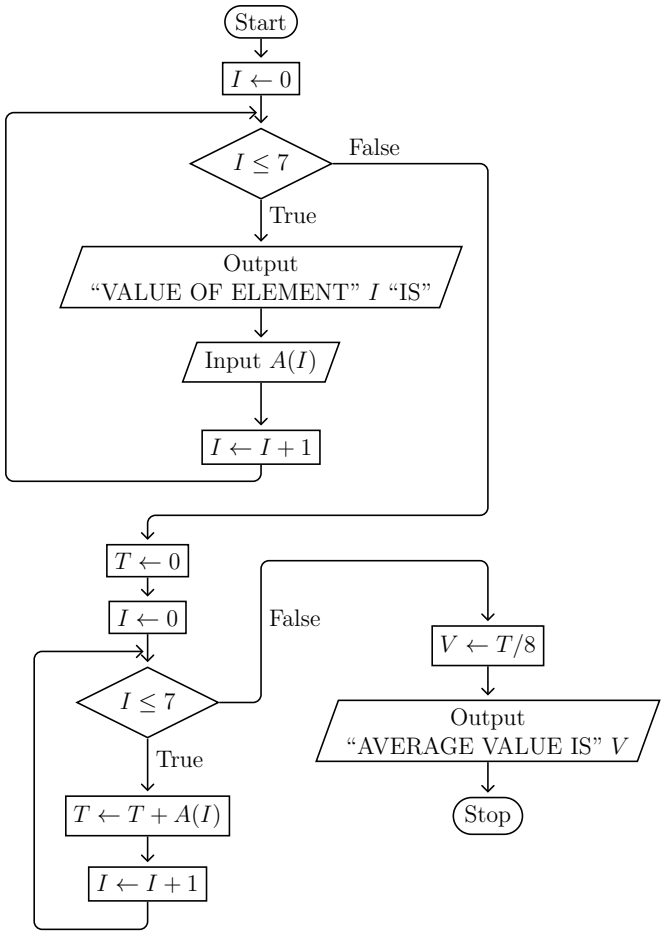


Figure 10.2: Compute Average Value Flowchart

```
10 REM   ARRAY DEMO
20 REM
30 REM A IS THE ARRAY OF 8 ELEMENTS
40 REM I IS AN INDEX FOR THE LOOP
50 REM T IS THE TOTAL VALUE OF ALL ELEMENTS
60 REM V IS THE AVERAGE VALUE OF ALL ELEMENTS
70 REM
80 DIM A(7)
90 REM GET THE ELEMENT VALUES
100 FOR I=0 TO 7
110 PRINT "VALUE OF ELEMENT";I;"IS ";
120 INPUT A(I)
130 NEXT I
140 REM COMPUTE AVERAGE VALUE OF THE ELEMENT VALUES
150 LET T=0
160 FOR I=0 TO 7
170 LET T=T+A(I)
180 NEXT I
190 LET V=T/8
200 PRINT "AVERAGE VALUE IS:";V
210 END
```

Figure 10.3: Compute Average Value

```
VALUE OF ELEMENT 0 IS ? 10
VALUE OF ELEMENT 1 IS ? 20
VALUE OF ELEMENT 2 IS ? 0
VALUE OF ELEMENT 3 IS ? 5
VALUE OF ELEMENT 4 IS ? 15
VALUE OF ELEMENT 5 IS ? 7
VALUE OF ELEMENT 6 IS ? 13
VALUE OF ELEMENT 7 IS ? 10
AVERAGE VALUE IS: 10
```

Figure 10.4: Compute Average Value Program Output

Some people do not like using subscripts that start with zero, and prefer to think of arrays as starting with subscript 1. Computer science people call arrays that begin at index 1 *one-based arrays*. In ECMA-55 Minimal BASIC you can request that arrays start on 1 by using the special `OPTION BASE 1` statement. If you do this, please remember that for `DIM A(100)`, the subscripts will go from $1, 2, \ldots, 99, 100$ for a total of 100 elements. By default the subscripts would instead go from $0, 1, \ldots, 99, 100$ for a total of 101 elements. If the `OPTION BASE 1` statement is used, it must precede all `DIM` statements, and it will apply to all arrays in the program. When working with arrays, you must always keep in mind the actual memory layout of your arrays. Figure 10.5 shows the same data as in Figure 10.1 on page 79, but stored in a one-based array.

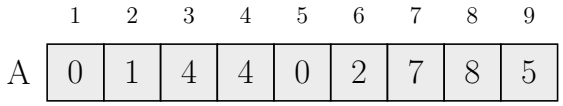


Figure 10.5: One-based Array A with 9 elements

10.2 Maximum Value

Let's look at another example program. Often a program must find the maximum value in a list of elements stored in an array. Putting data into the array is identical to the program described by the flowchart in figure 10.2 on page 81. Once again the program will then loop over all the elements to access each one. However, instead of adding each element value to a total, the element value will be compared to the current highest value. Since when we start looking there is no highest value, the program just chooses the first element as the highest value. It stores that value into the M variable, and stores the first index in the J variable. It then checks the rest of the elements one at a time and if any of them are larger than the current highest value stored in M, then M is updated to hold that new high value. The variable J is also updated to hold the index of that new highest value. After all elements have been examined, M will hold the highest element value of all the elements in the array, and J will hold the index where that value was found. A flowchart describing this algorithm is shown in figure 10.6 on page 84. This flowchart uses a one-based array, so we will need to use an `OPTION BASE 1` statement in our program.

The subscripts start at 1 in the program shown in figure 10.7 on page 86. This occurs because of the `OPTION BASE 1` on line 80. The `DIM(8)` on line 90 means there are 8 elements, with indices ranging from 1 through 8. The argument to `DIM`, which must be an unsigned integer value and not a variable or arithmetic expression, specifies not the size of the array, but the maximum possible subscript or index that you can use. To know the number of elements in an array, you

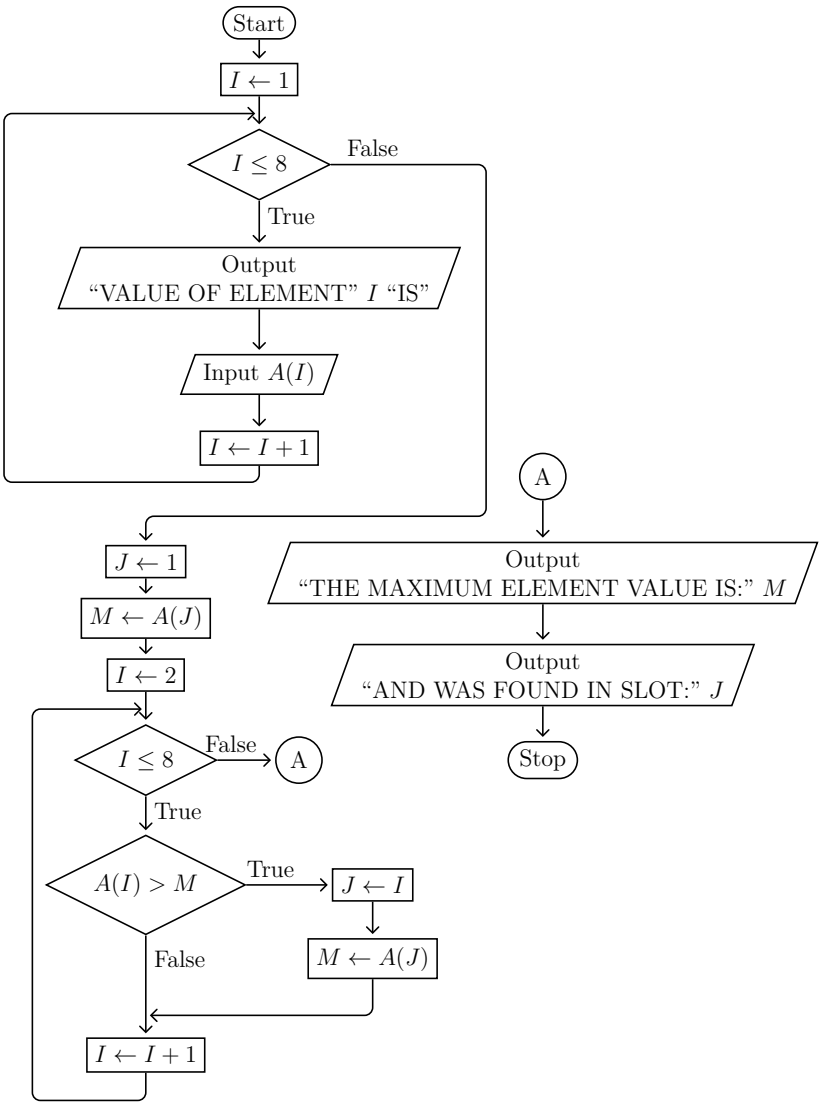


Figure 10.6: Find Maximum Value Flowchart

must know both the value specified in the `DIM` statement and whether the array is zero-based or one-based.

The corresponding ECMA-55 Minimal BASIC program implementing the flowchart shown in figure 10.6 on page 84 is shown in figure 10.7 on page 86. This program uses the `OPTION BASE 1` statement on line 80 to tell BASIC that one-based arrays will be used. For programs that expect a zero-based arrays, you can use `OPTION BASE 0` instead. It is not possible to mix both zero-based and one-based arrays in a single program. The `OPTION BASE` statement must occur before any `DIM` statements, and will determine the minimum index for all arrays in the program. Only 0 or 1 can be specified as arguments to `OPTION BASE`. You cannot have multiple `OPTION BASE` statements in the same program. The ECMA-55 Minimal BASIC standard specifies zero-based arrays are the default, but you should always include an `OPTION BASE` statement in any program that uses arrays to ensure your program will work correctly with any language translator (compiler or interpreter). Even when it is not absolutely required, you should still use an `OPTION BASE` statement because it documents what behavior the program expects.

The array data is loaded on lines 100 through 140. Then the first guesses for J and M are made on lines 160 and 170. The second loop stars on index 2 since `A(1)` was already used to initialize M. The second loop uses lines 180 through 220 and will find the maximum value and the location of that maximum value. Note that the condition on line 190 is opposite of that shown in the flowchart. This is because we have to jump over the code on lines 200 and 210 when `A(I)` is not larger than the current maximum value stored in M.

A sample run of the program from figure 10.7 on page 86 is shown in figure 10.8 on page 86. What happens if two elements tie for the maximum value? Will J be the index of the first occurrence or the second? You should type in and run the program and investigate these questions until you learn not only what the answer is, but why the algorithm works the way it does.

```
10 REM    FIND MAXIMUM VALUE IN A LIST
20 REM
30 REM A IS THE ARRAY OF 8 ELEMENTS
40 REM I IS AN INDEX FOR THE LOOP
50 REM M IS THE MAXIMUM VALUE OF ALL ELEMENTS
60 REM J IS THE INDEX WHERE THE MAXIMUM WAS FOUND
70 REM
80 OPTION BASE 1
90 DIM A(8)
100 REM GET THE ELEMENT VALUES
110 FOR I=1 TO 8
120 PRINT "VALUE OF ELEMENT";I;"IS ";
130 INPUT A(I)
140 NEXT I
150 REM FIND THE MAXIMUM VALUE
160 LET J=1
170 LET M=A(J)
180 FOR I=2 TO 8
190 IF A(I)<=M THEN 220
200 LET J=I
210 LET M=A(J)
220 NEXT I
230 PRINT "THE MAXIMUM ELEMENT VALUE IS:";M
240 PRINT "AND WAS FOUND IN SLOT:";J
250 END
```

Figure 10.7: Find Maximum Value

```
VALUE OF ELEMENT 1 IS ? -99
VALUE OF ELEMENT 2 IS ? 99
VALUE OF ELEMENT 3 IS ? 50
VALUE OF ELEMENT 4 IS ? 10
VALUE OF ELEMENT 5 IS ? -10
VALUE OF ELEMENT 6 IS ? 30
VALUE OF ELEMENT 7 IS ? 300
VALUE OF ELEMENT 8 IS ? 299
THE MAXIMUM ELEMENT VALUE IS: 300
AND WAS FOUND IN SLOT: 7
```

Figure 10.8: Find Maximum Value Program Output

The two example programs shown in this chapter are typical array processing programs. The program to compute the average uses all elements in a summation and then uses that as part of a formula. The program to find the maximum value inspects each element for some characteristic until it finds the required value.

10.3 More About Subscripts

Earlier it was stated that array subscripts must be numeric values. Since every numeric expression in ECMA-55 Minimal BASIC is a floating point value, you may be curious about what happens if you use a non-integral subscript value. The standard specifies that the expressions are rounded to the nearest integer, and the resulting integer value is used. However, it is bad style to use non-integral values, and it can make the program very, very hard to debug. This also means that several different subscript values can map to the same element if you use subscript values that are not integers. For instance, A(1.1), A(1.2), and A(1.3.14159) will all use A(1)'s element. A(1.5), A(1.9), A(1.91), and A(2.01) will all use A(2)'s element.

Consider the program and its output shown in figure 10.9. Two non-intuitive things have occurred:

1. The loop body was only executed twice, not three times as expected.
2. A(1.1) is not 1.1 as expected.

```
$cat DEMO.BAS
10 OPTION BASE 0
20 DIM A(4)
30 FOR X=1.1 TO 1.3 STEP .1
40 LET A(X)=X
50 NEXT X
60 FOR X=1.1 TO 1.3 STEP .1
70 PRINT "A(";X;")=";A(X)
80 NEXT X
90 END
$./BASICC DEMO.BAS
$./DEMO
A( 1.1 )= 1.2
A( 1.2 )= 1.2
$
```

Figure 10.9: Use of non-integral subscripts

The loop only executes twice because of the imprecision of floating point math. First, 1.1 cannot exactly be represented in IEEE-754 floating point. Then the code is adding .1, which is not exactly representable in IEEE-754 floating point either.

This addition occurs multiple times and involves a bit of error each time, and those errors are cumulative. The result of adding .1 to 1.1, and then adding .1 to that result, yields something that is not quite less than or equal to 1.3, so the third iteration does not occur. This is counter-intuitive since humans use base 10 math and these numbers are easy to work with. The IEEE-754 doubles are powers of 2, not powers of 10, and cannot exactly represent numbers that are not powers of 2. Usually the errors are small enough we do not notice, but in this case we get unexpected behavior. This is one reason why testing programs before using them for real work is essential.

Understanding the array output is a bit easier. Int the first iteration of the loop body, A(1.1) is assigned 1.1. That really stores 1.1 in A(1). In the second iteration of the loop body, A(1.2) is assigned 1.2. That really stores 1.2 in A(1). Yes, that is the same A(1) used earlier, so the 1.1 that was stored there is overwritten with the new value 1.2. In the second loop both print statements really print A(1), so they print the same A(1) value. Since the last value written to A(1) was 1.2, both lines print a value of 1.2 and neither shows the 1.1 value. Adding lines to show the values of X might help. Using `BASICCW`, which gives wide output, we can see what happened easily in figure 10.10.

```
$cat DEMO.BAS
10 OPTION BASE 0
20 DIM A(4)
30 FOR X=1.1 TO 1.3 STEP .1
35 PRINT X
40 LET A(X)=X
50 NEXT X
55 PRINT X
60 FOR X=1.1 TO 1.3 STEP .1
70 PRINT "A(";X;")=";A(X)
80 NEXT X
90 END
$./BASICCW DEMO.BAS
$./DEMO
 1.1
 1.2000000000000002
 1.3000000000000003
A( 1.1 )= 1.2000000000000002
A( 1.2000000000000002 )= 1.2000000000000002
$
```

Figure 10.10: Use of non-integral subscripts2

After two iterations, X is 1.3000000000000003, which is not less than or equal to 1.3, so the third iteration never occurs. This program is wrong and will never work the way it is written, even though logically it is clear. How can we write a program that does what we intended? Several approaches are possible. I show a simple one in Figure 10.11 on page 89. This programs gives the output that was intended, but using only integral loop and array indices.

```
$cat DEMO2.BAS
10 OPTION BASE 0
20 DIM A(2)
30 FOR X=11 TO 13 STEP 1
40 LET A(X-11)=X/10
50 NEXT X
60 FOR X=11 TO 13 STEP 1
70 PRINT "LOGICAL A(";X/10;")=";A(X-11)
80 NEXT X
90 END
$./BASICC DEMO2.BAS
$./DEMO2
LOGICAL A( 1.1 )= 1.1
LOGICAL A( 1.2 )= 1.2
LOGICAL A( 1.3 )= 1.3
$
```

Figure 10.11: Using logical non-integral subscripts

Exercises

1. Modify the flowchart shown in figure 10.6 on page 84 to find the minimum value in an array.

2. Modify the program shown in figure 10.7 on page 86 to match the flowchart in the previous question and test it to verify it really does find the minimum value in an array.

Chapter 11

Including Data Within A Program

Often a program needs some constant data included within the program itself. This is especially true for ECMA-55 Minimal BASIC since it has no capability for reading files. Fortunately BASIC has special support for including data within the program using the **DATA** and **READ** statements. The **DATA** statement is used to include the actual data within the program. A **DATA** statement begins with a line number like all BASIC statements. It uses the **DATA** keyword followed by a comma-delimited list of one or more data items. The **DATA** statement can occur anywhere within the program, but traditionally most people include them at the end of the program just before the **END** statement. Including the data is easy, but how do you access it?

BASIC includes a special **READ** statement for this. The syntax of the **READ** statement is similar to the **INPUT** statement you already know. After the initial line number, the **READ** keyword is used, and this is followed by a list of zero or more comma-delimited variable names to load with data. The variables are loaded in left-to-right order as you would expect. The **READ** statement behavior is very similar to the **INPUT** statement behavior, but instead of getting data from the keyboard, it gets data from the built-in data specified in the **DATA** statements of the program. Logically, the program is scanned when you start to run it in order to find all of the **DATA** statements. Each one is processed in turn and all of the data items on that line are appended to the data items list. If a program has no **DATA** statements, then the data items list is empty. If the program has many data statements, all that data is read in order to create one large list of data items.

11.1 A simple example using READ and DATA

The first **READ** statement will begin reading from the beginning of the data item list. The program will remember what has been read, and on the next **READ** new data from the list will be read. If a **READ** statement is used but no more data items exist on the list to read, a fatal error will occur, so the programmer must ensure that the program never reads more data items than are available. The example program shown in figure 11.2 on page 93 should help you understand how the **READ**

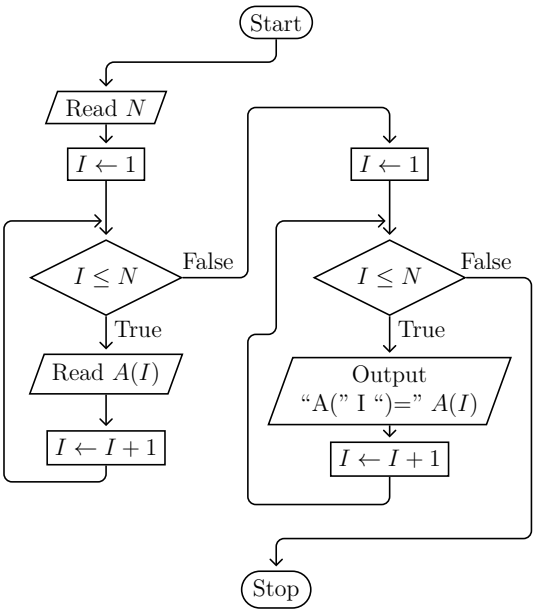


Figure 11.1: Load Array With Data Flowchart

and **DATA** statements work together to allow the program to include data within itself. To help you understand the program, a flowchart for the program is provided in figure 11.1.

After examining the sample program shown in figure 11.2 on page 93, you should type it in and run it. The output will be as shown in figure 11.3 on page 93. The **DATA** statements are on lines 230, 250 and 260, and all of their data items are merged into one long list of data items. Line 130 reads the first data item into the variable **N**, and that value is the number of data items available. Your array **A** must have room for all of the elements. Since this sample uses 15 data items and the array **A** has 31 elements, the data will fit. The **READ** on line 150 is executed 15 times, once for each of the items in the data list after the initial count value which was already read into the variable **N** on line 130. As you can see in the program output, all of the data items were indeed added to the internal data item list in the order in which they appeared in the program. By using **DATA** and **READ**, it is possible to include data within your program so that the user does not have to type it all in from the keyboard. The other alternative would be to add 16 **LET** statements, one for each array element, and one for the count variable **N**. Instead it is possible to do that with just 4 lines. Another advantage of using **READ** and **DATA** is that if you change the data, you do not have to change the main program logic. You must be careful to ensure the data item value specified on line 230 matches the number of data items, and that the specified size of array **A** is large enough to contain that many data items.

In some programs, especially games, you might want to reset the game. To do that, you would need to read all the data again. There is a special **RESTORE** statement

```
10 REM    DATA STATEMENT EXAMPLE
20 REM    WRITTEN BY JOHN GATEWOOD HAM
30 REM    LAST MODIFIED 2016/02/23 AT 16:46 ICT
40 REM
50 REM A   THIS IS THE ARRAY USED BY THIS PROGRAM
60 REM     AND CAN HAVE A MAXIMUM OF 31 ELEMENTS
70 REM     WITH SUBSCRIPTS 0 .. 30
80 REM I   THIS IS USED AS THE INDEX FOR THE FOR LOOPS
90 REM     AND IS ALSO THE SUBSCRIPT USED WITH ARRAY A
100 REM N   CONTAINS NUMBER OF ELEMENTS USED IN ARRAY
110 DIM A(30)
120 REM LOAD ARRAY DATA
130 READ N
140 FOR I=1 TO N
150 READ A(I)
160 NEXT I
170 REM DISPLAY ARRAY CONTENTS
180 FOR I=1 TO N
190 PRINT "A(";I;") =";A(I)
200 NEXT I
210 STOP
220 REM NEXT LINE CONTAINS NUMBER OF DATA ITEMS
230 DATA 15
240 REM FOLLOWING LINES CONTAIN DATA FOR ARRAY
250 DATA 9,9,8,8,7,7,6,6,5,5
260 DATA 4,4,3,3,2
270 END
```

Figure 11.2: READ and DATA example

```
A( 1 ) = 9
A( 2 ) = 9
A( 3 ) = 8
A( 4 ) = 8
A( 5 ) = 7
A( 6 ) = 7
A( 7 ) = 6
A( 8 ) = 6
A( 9 ) = 5
A( 10 ) = 5
A( 11 ) = 4
A( 12 ) = 4
A( 13 ) = 3
A( 14 ) = 3
A( 15 ) = 2
```

Figure 11.3: READ and DATA Program Output

to help you do that. The **RESTORE** statement will tell the program that no data has been read yet. After a **RESTORE** statement, the next **READ** will read the first data item value in the program, not the next data item value on the internal list of item values as it usually would.

11.2 Three ways to read lists of data

In many programs, you will have lists of constant data which you need to read into variables. There are three common ways to read lists of constant data from **DATA** statements in BASIC:

1. Read until a special *sentinel* value is read.
2. Read the list length as the first data value.
3. Hard-code the list length.

Each style has strengths and weaknesses, and if you look at very many older BASIC programs, you are almost certainly going to see all of these methods used.

Let's look at an example program that shows all three common ways to read constant data from **DATA** statements. After examining the sample program shown in figure 11.4 on page 95, you should type it in and run it. The output will be as shown in figure 11.5 on page 96.

The first style of reading data occurs on lines 80 through 180. This uses a sentinel value of -999 to mark the end of the data. A *sentinel* value is a special value that cannot be present in the set of normal data values, but which does share the same data type as the normal data values. The second style of reading data occurs on lines 230 through 350. This reads the number of data values as the first data item and stores that value in the variable **C** with the **READ** statement on line 240. The third and final style of reading data occurs on lines 450 through 530, and has the number of items hard-coded in the **LET** statement on line 450. Some people would just omit line 450 and instead hard-code the value **4** instead of **C** in the **FOR** statement on line 470. Note well the use of **RESTORE** on line 360 to reset the data pointer to prepare for the third phase. This resets the internal data pointer back to the first element in the first **DATA** statement on line 550, so the **READ** on line 480 will begin reading from the first element of DATA in the program.

So when should you use which style? If you are sure the number of elements will never change, the last style with the hard-coded number of elements is the most efficient. However, if you decide to change the number of items in your data, you must check the program logic carefully to update *all* of the hard-coded list length values. If you often quickly add or remove elements, the first style with a sentinel is easier to use, but it does require that a sentinel value exists which will never be

```
10  REM         DEMONSTRATE READ, DATA, AND RESTORE
20  REM
30  REM IF YOU DO NOT KNOW HOW MANY ITEMS TO READ, YOU NEED TO HAVE
40  REM A SENTINEL VALUE AT THE END OF THE LIST WITH A VALUE THAT
50  REM ABSOLUTELY CANNOT OCCUR IN THE SET OF NORMAL VALUES YOU READ.
60  REM IN THIS CASE, -999 IS THE SENTINEL VALUE.
70  REM
80  LET C=0
90  READ A
100 IF A=(-999) THEN 170
110 IF A>=0 THEN 140
120 PRINT "YOU READ ";A
130 GOTO 150
140 PRINT "YOU READ";A
150 LET C=C+1
160 GOTO 90
170 PRINT "YOU READ";C;"ITEMS"
180 PRINT
190 REM
200 REM SOMETIMES THE LENGTH OF THE LIST IS IN THE DATA ITSELF AS THE
210 REM FIRST ITEM OF THE LIST.
220 REM
230 LET I=0
240 READ C
250 PRINT "YOU SHOULD READ";C;"ITEMS"
260 IF C=0 THEN 350
270 READ A
280 LET C=C-1
290 LET I=I+1
300 IF A>=0 THEN 330
310 PRINT "ITEM #";I;": ";A
320 GOTO 340
330 PRINT "ITEM #";I;":";A
340 GOTO 260
350 PRINT
360 RESTORE
370 REM
380 REM THE RESTORE STATEMENT RESETS THE READ STATEMENT'S LOCATION
390 REM FOR THE NEXT ITEM TO READ BACK TO THE BEGINNING OF THE
400 REM LIST OF ITEMS.
410 REM
420 REM IF YOU KNOW HOW MANY ITEMS TO READ, YOU CAN USE A FOR LOOP.
430 REM IN THIS CASE, WE KNOW THERE ARE C DATA VALUES TO READ.
440 REM
450 LET C=4
460 PRINT "YOU SHOULD READ";C;"ITEMS"
470 FOR I=1 TO C STEP 1
480 READ A
490 IF A>=0 THEN 520
500 PRINT "ITEM #";I;": ";A
510 GOTO 530
520 PRINT "ITEM #";I;":";A
530 NEXT I
540 STOP
550 DATA 1,2,3,-4,-999
560 DATA 4,1,2,3,-4
570 END
```

Figure 11.4: Example of reading a list of data three different ways

```
YOU READ 1
YOU READ 2
YOU READ 3
YOU READ -4
YOU READ 4 ITEMS

YOU SHOULD READ 4 ITEMS
ITEM # 1 : 1
ITEM # 2 : 2
ITEM # 3 : 3
ITEM # 4 : -4

YOU SHOULD READ 4 ITEMS
ITEM # 1 : 1
ITEM # 2 : 2
ITEM # 3 : 3
ITEM # 4 : -4
```

Figure 11.5: Output of program in figure 11.4

present in the normal data. The second style is a compromise that allows *any* value in the normal data, yet avoids hard-coding any list size in your loop code. It does require you to manually keep track of the number of data items when you update your program, but your main program code logic does not need any changes. If you are not sure which style to use, I recommend the second style with the list length as the first element of data.

Exercises

1. Create a flowchart for the program shown in figure 11.4 on page 95.
2. Modify the program shown in figure 11.2 on page 93 to find and display the largest value in the array after it has been loaded with data.
3. Add two more data item values to both lists in the program shown in figure 11.4 on page 95 and verify the program still works correctly.

Chapter 12

Sequential Search

If you have data stored in an array, you often will have a need to find an element in the array. This is called searching for an element. Many ways exist to search for elements, but the first one to learn is the simplest, and is called *sequential search*. The idea behind the sequential search is quite simple. Once you know the value you are looking for, look in the first element of the array and see if you find it. If you found it, you stop and report success. If not, move to the next element and see if that value matches. Continue this process until you find the value or you have checked every element in the array. If you have checked every element in the array and the value was not found, then you can report failure.

12.1 The Algorithm

A step-by-step diagram showing a search for the value 5 in array A with 9 elements is shown in figure 12.1 on page 100. To move through the array, the index variable I is used. The first row shows checking the first element, A_0, and that element is shaded. Since the value in that element is 0, not 5, the search has not succeeded yet. Since there are still more elements in the array, the search will continue by incrementing the index I and trying the test to check if $A_I = 5$ again. The second row shows that A_1 is 1, not 5, and there are still more elements in the array, so the search still has not succeeded yet. I is incremented and the search continues. The third row shows that A_2 is 4, not 5, and there are still more elements in the array, so the search still has not succeeded yet. I is incremented and the search continues. In fact, the search continues for the values of I from 3 through 7 without success. On the ninth and final element, A_8, the element value *is* 5, so the search finally succeeds in the last element. If the search had been for another value, say 9, that was not in the array, the search would still end after element 8 since when I was incremented to 9 it would be greater than the maximum subscript 8, but in that case the search would end with failure.

Sequential search is guaranteed to find the first occurrence of a value in the array if one exists. However, as the example showed, in the worst case every element in the array must be checked. This means that sequential search worst case will

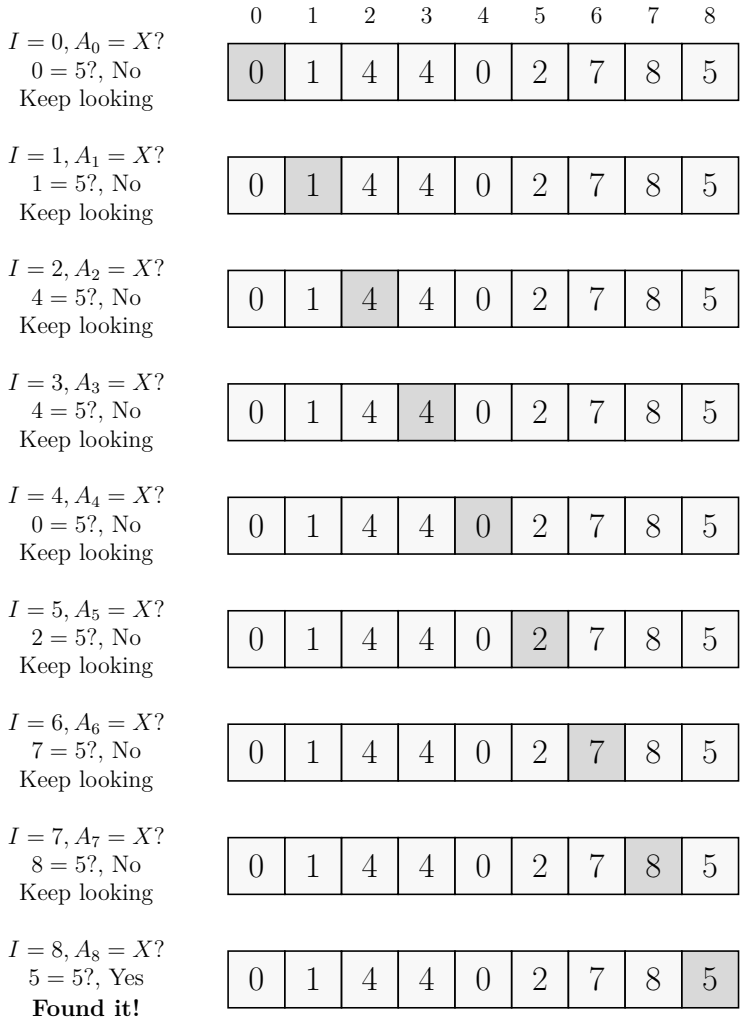


Figure 12.1: Sequential search for X=5 in array A

need $O(n)$[1] compares for an array with n elements. For small values of n it works well, but as n grows to larger values, sequential search can be quite slow. The big advantage of sequential search is that it is simple and easy to remember.

What happens if a search is tried for something that is not in the list? That will require checking every element, and then after the last element the program must stop looking and report that the value was not found in the list. As you can see in figure 12.2 on page 101, where we search for -1 which is **not** in the list, if the search is not successful the program still needs to check every element.

What if we search for a value that is in the list but **not** stored in the last element? To find out, we will see what happens when we search for the value 4. The value 4

[1] $f(n) = O(g(n))$ means that there are positive constants c and k, such that $0 \leq f(n) \leq c \times g(n)$ for all $n \geq k$. $O(n)$ is spoken as big-oh of n.

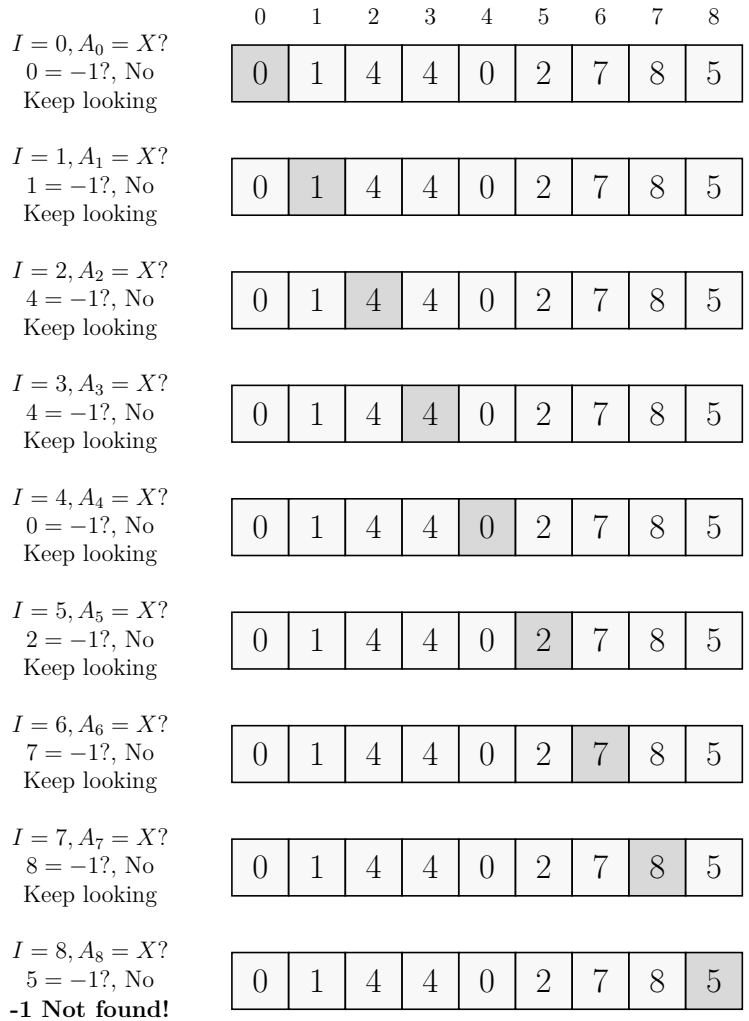


Figure 12.2: Sequential search for X=-1 in array A

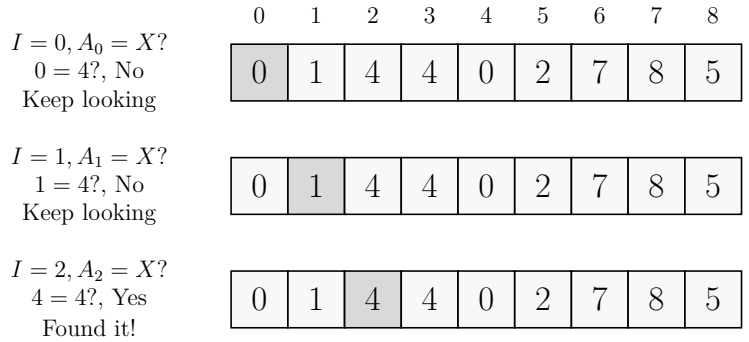


Figure 12.3: Sequential search for X=4 in array A

is in the list twice, once in element 2 and once in element 3. The computer science way to express the case when a value occurs more than once is to say the list contains *duplicate values*. So this example has duplicate values of 4 in elements 2 and 3. It also has duplicate values of 0 in elements 0 and 4. For sequential search, the search for a value of 4 would stop when the first matching value occurs, so when the value stored in element 2 is compared with a search value of 4, the search would stop and report success. This is shown in figure 12.3 on page 101. Element 3 would not be examined. That is why a sequential search is guaranteed to find the *first* occurrence of a value in the array if one exists.

So how can this idea actually be implemented? The first thing to consider is that in order to search an array for data, first data must be loaded into the array. Loading data into an array was explained in chapter 11, and that technique will be used in this chapter. This means a program to implement this search has two distinct phases: first loading the data, and then searching for that data in the array. These distinct phases are visible in the flowchart shown in figure 12.4 on page 103.

The flowchart has two lightly shaded areas, one for the *load data* code and one for the *sequential search* code. The code between them is used to get the user to enter the value for which to search. The code after the *sequential search* is used to report the results of the search.

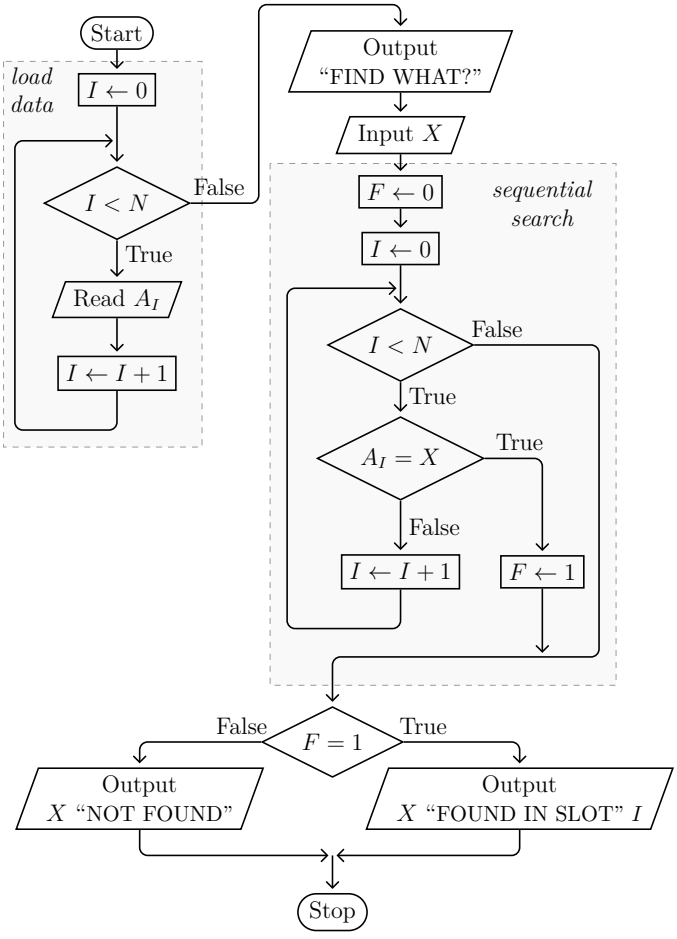


Figure 12.4: Sequential Search Flowchart for N elements

12.2 Implementation in ECMA-55 Minimal BASIC

The BASIC code that corresponds to the *load data* box is made of several parts and is shown in figure 12.5 on page 105. The `DIM` statement on line 80 creates the array A with room for 20 elements. The lines 100 through 120 contain the `FOR` loop that actually loads the array, and line 290 contains the `DATA` statement with the data values that get loaded. This technique is explained in detail in chapter 11.

The BASIC code that corresponds to the *sequential search* box is on lines 170 through 220. This also uses the same style of pre-test loop implemented with a `FOR` statement as the *load data* code. However, the loop body is different. It contains a test that checks to see if the current array element $A_I = X$, and if it is the flag F is set to 1 to indicate that the value was found and then the code exits the loop. If the test is false, however, we increment I and keep looking. If X is found the loop exits and F will be 1 and I will be the index where X was found. If X is not found, the loop will exit when I is 20 and F will be 0.

Lines 240 through 270 report the results of the search, and line 280 causes the program to terminate using the `STOP` statement. Unlike the special `END` statement which must exist exactly once as the last line of the program, the `STOP` statement may be used as many times as you wish in the program. Any time the program reaches a `STOP` statement or the `END` statement, the program will terminate.

```
10 REM      SEQUENTIAL SEARCH DEMO
20 REM
30 REM      A IS ARRAY TO HOLD THE DATA ITEMS
40 REM      I IS THE LOOP INDEX VARIABLE
50 REM      X HOLDS THE VALUE WE SEEK
60 REM      F IS A FLAG, 0 MEANS NOT FOUND, 1 MEANS FOUND
70 REM
80 DIM A(19)
90 REM      READ DATA INTO ARRAY A
100 FOR I=0 TO 19
110 READ A(I)
120 NEXT I
130 REM      GET VALUE FOR WHICH TO SEARCH
140 PRINT "FIND WHAT";
150 INPUT X
160 REM      DO SEQUENTIAL SEARCH
170 LET F=0
180 FOR I=0 TO 19
190 IF A(I)<>X THEN 220
200 LET F=1
210 GOTO 240
220 NEXT I
230 REM      REPORT RESULTS
240 IF F=1 THEN 270
250 PRINT X;"NOT FOUND"
260 GOTO 280
270 PRINT X;"FOUND IN SLOT";I
280 STOP
290 DATA 21,85,80,14,60,76,87,49,78,81,96,25,17,22,13,91,23,62,5,57
300 END
```

Figure 12.5: Sequential Search Program

Exercises

1. What would you need to change in the program if you want to have a list with 30 elements?

2. Some lists contain duplicate values. Modify the flowchart so that instead of using F as a Boolean flag, F becomes a count of how many copies of X that were found. In this case you would report the *last* copy of X found.

3. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

4. Continuing with the flowchart in the first question, add a new array B that will contain all of the indices of X values found. If X is 7 and you found 3 copies, $B(0)$ contains the index of the first copy, $B(1)$ contains the index of the second copy, and $B(2)$ contains the index of the last copy. You will need a scalar $B0$ to remember how many elements in B are used.

5. Write the ECMA-55 Minimal BASIC program for the flowchart in the previous question.

6. What happens if the list contains duplicate values? Update the flowchart and then the program to ensure that all values in the list matching the search value are shown.

Chapter 13

Subroutines

In many programs, you will need to do the same thing many times. We have already learned one way to do this with pre-test and post-test loops. However, sometimes we need something more powerful. We want to be able to write some code to do something that allows us to use that code again and again whenever we need it, but not necessarily within a loop. The general concept to support this in computer programming is called the *subroutine*.[1] The ECMA-55 Minimal BASIC language has support for implementing subroutines that uses two special statements. The `GOSUB` statement will tell the program to run a subroutine, and the `RETURN` statement will tell the subroutine to exit. Unlike modern languages that use names for subroutines, ECMA-55 Minimal BASIC uses the line number of the first line of the subroutine to specify what subroutine to run. This line number follows the `GOSUB` keyword in the same style that `GOTO` uses. When a subroutine is invoked with the `GOSUB` keyword, the program will jump to the specified line number and then run normally until a `RETURN` statement occurs. When the `RETURN` statement executes, it will jump back to the line after the `GOSUB` statement that was used to invoke the subroutine. It is an error to use `RETURN` without using `GOSUB` first to enter the subroutine. Subroutines can call other subroutines, and the ECMA-55 Minimal BASIC runtime will remember the details so that the `RETURN` statements work correctly.[2]

To use subroutines in a flowchart, we need to learn some new symbols. When a program runs a subroutine, there are some traditional computer science names for that. One verb used for this action is call, and people will say they *call* a subroutine. Another verb used is invoke, and people say they will *invoke* a subroutine. This book will use the *call* verb for this concept. The symbol used when you want to call a subroutine is a rectangle where the left and right sides have double lines, and is called the *predefined process symbol*. The start of the subroutine uses terminal symbols at the beginning and end of the subroutine. The terminal at the end of

[1] Many languages like C have functions, which are subroutines that return a value. Some, like Pascal, have procedures, which are subroutines that do not return a value.

[2] Using line numbers as an entry points and permitting multiple return statements allows multiple entry and exit points to a subroutine, but doing that is considered bad style by many programmers since it makes the code more difficult to read and maintain.

the subroutine will have the word "Return" inside of it. The terminal at the start of the subroutine will have some short unique identifier name for the subroutine. Like many programming concepts, the subroutine idea is harder to describe in words than in a picture, so please examine the flowchart shown in figure 13.1 and figure 13.2 on page 109.

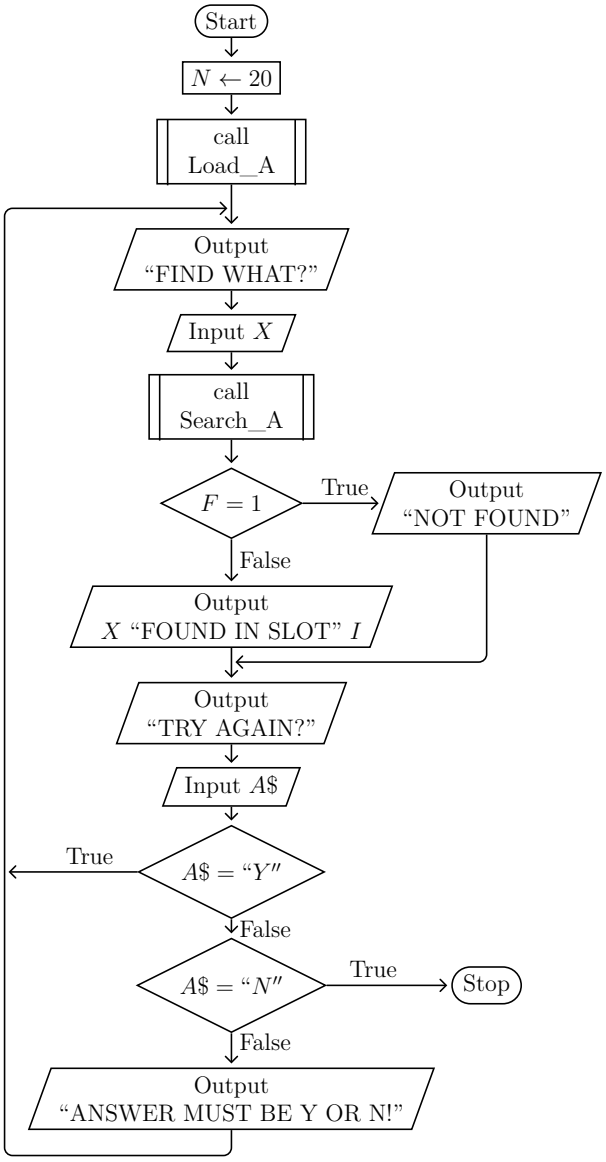


Figure 13.1: Subroutine Example Flowchart (1 of 2)

The main program logic is shown in figure 13.1 and it uses the new predefined process symbol twice. The first time it is used to call the "Load_A" subroutine, and the second time it is used to call the "Search_A" subroutine. Those two subroutines are shown in figure 13.2 on page 109. The structure of this flowchart is

easier to read, since the main logic hides the details of the array load and sequential search. You could recycle the main program logic but change how data is loaded by just replacing the "Load_A" subroutine. You can change to a different kind of search by just replacing the "Search_A" routine. All good large programs use subroutines to make the structure of the program more modular.

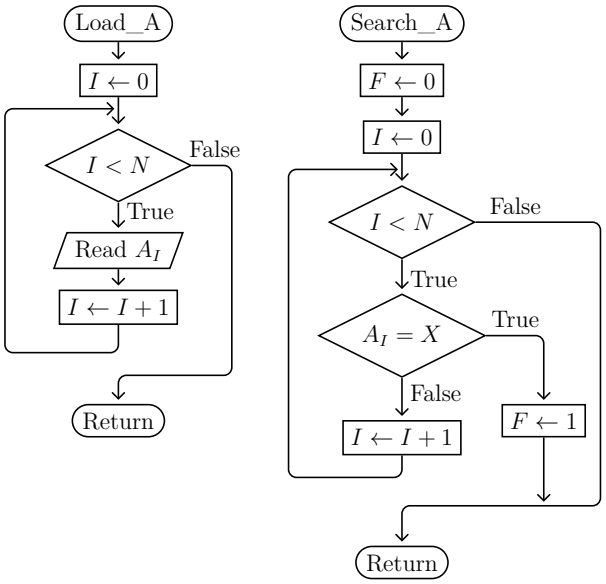


Figure 13.2: Subroutine Example Flowchart (2 of 2)

Now it is time to examine the ECMA-55 Minimal BASIC source code in figure 13.3 on page 110 which implements the flowcharts. The BASIC code for the main program logic is on lines 10 through 340. The in-program data is on line 630. The BASIC code for the "Load_A" subroutine is on lines 380 through 450. The BASIC code for the "Search_A" subroutine is on lines 470 through 600. Note that the `STOP` statement on line 340 is needed to ensure the main program stops instead of accidentally falling through to the subroutines. You could use `GOTO 650` instead of `STOP`, but then if you ever change the program making it longer, by adding more data for instance, then you would have to remember to update that `GOTO` line number. By using `STOP`, no line number is used so it always works even if the line numbers change, and this is why it is always best to use a `STOP` statement at the end of your main program logic if your program has any subroutines.

One other new technique is used in this sample program on line 90. That line specifies `OPTION BASE 0` so that we get zero-based arrays. However, those are the default anyway, so why include line 90? This is done in case the program is *ported* to run on another *dialect* of BASIC. BASIC has hundreds of dialects, each slightly different. This is very much like English. Most people born in Australia, England, Scotland, Ireland, Canada, and the USA speak English, but they speak different

```
10 REM      SEQUENTIAL SEARCH DEMO WITH SUBROUTINES
20 REM
30 REM      A IS ARRAY TO HOLD THE DATA ITEMS
40 REM      I IS THE LOOP INDEX VARIABLE
50 REM      X HOLDS THE VALUE WE SEEK
60 REM      F IS A FLAG, 0 MEANS NOT FOUND, 1 MEANS FOUND
70 REM      N IS NUMBER OF ELEMENTS IN A
80 REM
90 OPTION BASE 0
100 DIM A(19)
110 REM
120 REM *************** MAIN ******************
130 REM
140 REM      READ DATA INTO ARRAY A
150 LET N=20
160 GOSUB 380
170 REM      GET VALUE FOR WHICH TO SEARCH
180 PRINT "FIND WHAT";
190 INPUT X
200 REM      DO SEQUENTIAL SEARCH
210 GOSUB 470
220 REM      REPORT RESULTS
230 IF F=1 THEN 260
240 PRINT X;"NOT FOUND"
250 GOTO 270
260 PRINT X;"FOUND IN SLOT";I
270 REM      TRY AGAIN?
280 PRINT "TRY AGAIN";
290 INPUT A$
300 IF A$="Y" THEN 170
310 IF A$="N" THEN 340
320 PRINT "ANSWER MUST BE Y OR N!"
330 GOTO 280
340 STOP
350 REM
360 REM *************** SUBROUTINES ******************
370 REM
380 REM SUBROUTINE TO LOAD DATA FROM DATA STATEMENTS INTO A
390 REM INPUT N NUMBER OF ELEMENTS
400 REM OUTPUT A(), ARRAY WITH N ELEMENTS
410 REM
420 FOR I=0 TO N-1
430 READ A(I)
440 NEXT I
450 RETURN
460 REM
470 REM SUBROUTINE TO DO SEQUENTIAL SEARCH FOR X IN A
480 REM INPUT N NUMBER OF ELEMENTS
490 REM INPUT A(), ARRAY WITH N ELEMENTS
500 REM INPUT X, ELEMENT VALUE TO SEARCH FOR
510 REM OUTPUT F, 0 MEANS NOT FOUND, F=1 MEANS FOUND
520 REM OUTPUT I, INDEX OF X IN A() IF F=1, N OTHERWISE
530 REM
540 LET F=0
550 FOR I=0 TO N-1
560 IF A(I)<>X THEN 590
570 LET F=1
580 GOTO 600
590 NEXT I
600 RETURN
610 REM
620 REM ****************** DATA ********************
630 DATA 21,85,80,14,60,76,87,49,78,81,96,25,17,22,13,91,23,62,5,57
640 REM ****************** END ********************
650 END
```

Figure 13.3: Subroutine Example Program

dialects of English. For example, a biscuit means a kind of bread product to people from the USA but it is a type of a cookie to people in England. So some BASIC implementations are zero-based, but not all of them are. Adding the `OPTION BASE 0` explicitly tells BASIC we expect to use zero-based arrays. When a program is modified to run in a different environment, such as on another operating system, this is called *porting* the program. If you need to update a program for a different dialect of the language, for instance updating an old K&R C program to use C89, this is also *porting* the code. So by using the `OPTION BASE 0` statement, we make the code more *portable* (easier to port) since it lets anyone who reads the source code know we expect zero-based arrays and if their BASIC uses one-based arrays and does not support the `OPTION BASE 0` statement, then they will need to modify the program and check all the subscripts very carefully.

Exercises

1. Rewrite the flowchart in figure 11.1 on page 92 using the ideas from this chapter so that the array load and array print loops are each in their own subroutines.

2. Write a BASIC program to implement the flowchart you created as an answer to the previous question. You might want to look at figure 11.2 on page 93 for inspiration.

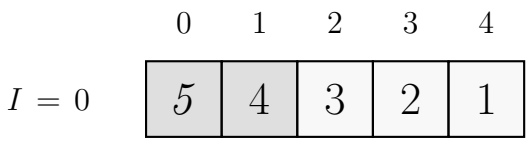
Chapter 14

Bubble Sort

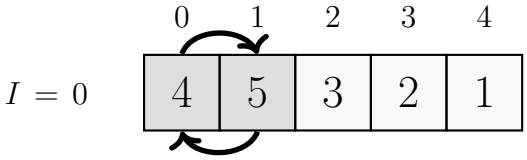
This chapter describes a simple and effective sorting technique for small, integer-indexed arrays like those used in ECMA-55 Minimal BASIC. This sort will put all the data values into ascending order, with the lowest value first and the highest value last. The sorting technique is called the *bubble sort* since the larger values bubble up to the top of the array, where the top of the array is the end of the array with the highest index. The way this sort works is by moving the largest remaining element to the end of the list every iteration of the inner loop by comparing adjacent values and swapping values when they are out of place, moving from index 0 to index $N - 1$, for an array with N elements. After each time through the array, one more element is moved into the correct position. Of course, if we are lucky and the element was already in the correct position then it does not need to actually be moved. To be certain all elements are in the correct position at the end of the sort, the array data must be processed $N - 1$ times.

14.1 The Algorithm

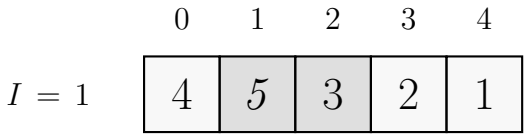
The best way to learn how the algorithm works is by examining a series of diagrams. The example that follows uses a zero-based array A with five elements which initially are in reverse order, which is the worst case for an ascending order bubble sort. Each time the outer loop iterates, the inner loop will run to ensure the largest element remaining is moved into place. For the first iteration of the inner loop during the first iteration of the outer loop, this means that the largest element in the array will be moved into the last element of the array. The sort begins with the array values already loaded into array A and the variable N initialized to the number of elements, 5.



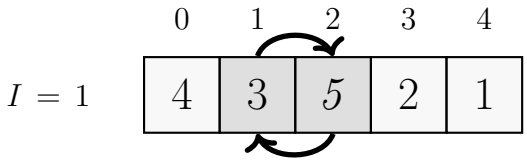
Is $A_I > A_{I+1}$? Is $A_0 > A_1$? Is $5 > 4$? Yes so the values are in the wrong order.



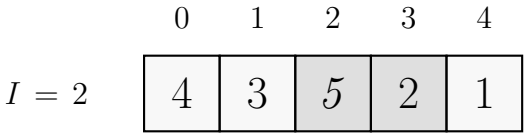
So the program must swap the values stored in A_0 and A_1 as shown. Then I is incremented and the next pair of values must be examined.



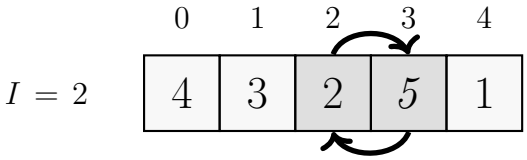
Is $A_I > A_{I+1}$? Is $A_1 > A_2$? Is $5 > 3$? Yes the values are in the wrong order.



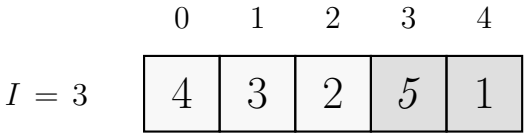
So the program must swap the values stored in A_1 and A_2 as shown. Then I is incremented and the next pair of values must be examined.



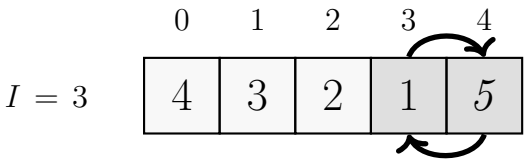
Is $A_I > A_{I+1}$? Is $A_2 > A_3$? Is $5 > 2$? Yes the values are in the wrong order.



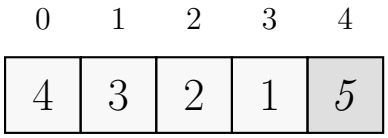
So the program must swap the values stored in A_2 and A_3 as shown. Then I is incremented and the next pair of values must be examined.



Is $A_I > A_{I+1}$? Is $A_3 > A_4$? Is $5 > 1$? Yes the values are in the wrong order.

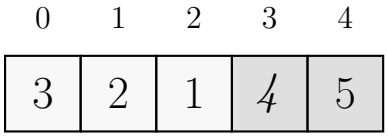


So the program must swap the values stored in A_3 and A_4 as shown.

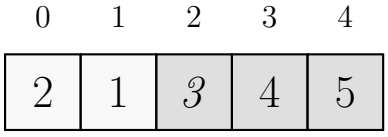


Value 5 is the largest value and will never move again now that it has been moved into its proper place in the array, that is, in the last element A_4.

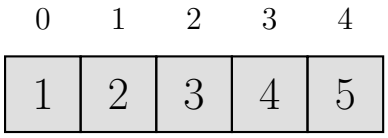
That completes the inner loop for the first iteration of the outer loop. The second iteration of the outer loop will move 4 into position, yielding the following:



The third iteration of the outer loop will move 3 into position, yielding the following:



The fourth and final iteration of the outer loop will move 2 into position, and as a by-product the 1 will fall into place too, yielding the following:



One complete iteration of the inner loop moved largest value, 5, into the last element of the array, A_4, and it also reduced the number of elements that need to be examined in the future by one. The next inner loop only needed to examine the first four elements. The inner loop for the second iteration moves the 4 value into the fourth element, A_3. Each iteration of the outer loop moves one more element into its final position and reduces the number of elements the inner loop must examine by one. The last outer loop iteration puts the second element into place, and the first element is then in the correct location as a by-product of that. After

the outer loop has caused the inner loop to run $N - 1$ times, all elements will have been forced into the correct places in the array. In this example, that is $5 - 1 = 4$ times.

Since there are two loops with the outer loop needing $n - 1$ iterations and the inner loop in the worst case needing to examine $n - 1$ elements, that yields a high estimate of $O((n-1)(n-1))$, or $O(n^2 - 2n + 1)$ for the bubble sort algorithm. The actual number of iterations for the inner loop will keep shrinking as more and more elements are put in place at the end of the array and the number of remaining elements that must be sorted in the next pass decreases. Anyway, the n^2 term dominates, so this is called an $O(n^2)$ algorithm in the number of compares and swaps in the worst case.

As coded, the average case performance is equal to the worst case. However, the bubble sort is simple to code and simple to remember, and on modern hardware the performance is adequate for short lists of up to 100 scalars. As n increases, the runtime increases quadratically, making this algorithm bad for sorting large lists.

How can we use this algorithm in our programs? Well, first a flowchart must be created that shows the step-by-step details of how the algorithm works. A suitable flowchart is provided in figure 14.1 on page 117.

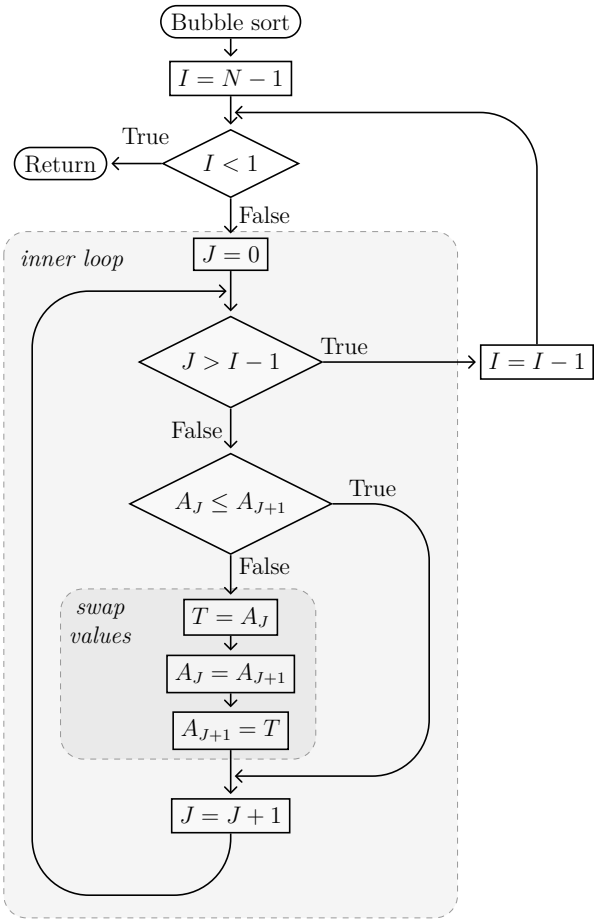


Figure 14.1: Bubble sort flowchart for N elements

14.2 Implementation in ECMA-55 Minimal BASIC

So how can this algorithm be implemented in ECMA-55 Minimal BASIC? The flowchart only shows the actual sorting algorithm, since previous chapters have shown several ways to load and display the data stored in an array. A program which loads random data values into an array, then displays the unsorted list, then sorts the values, and finally displays the sorted list is shown in figure 14.2 on page 119.

The actual bubble sort occurs in the subroutine that begins on line `260`. For elements which are determined to be out of order by the test on line `350`, the swap occurs on lines `360` to `380`. The inner loop on lines `340` to `390` will ensure the largest value in the unsorted part of the array is in the correct position during each iteration. The outer countdown loop is used to ensure both that the inner loop occurs $N - 1$ times, and that the inner loop only processes the values in the unsorted part of the list.

The unsorted part of the list keeps getting shorter by one element upon each iteration of the outer loop, since the largest value in the unsorted part of the list moves to the end of the unsorted list. The inner loop executes one less time for each iteration of the outer loop, because the number of elements remaining in the unsorted part of the array decreases by one for each iteration of the outer loop.

The subroutine to print the values in the array **`A`** is on lines `180` to `250`. Notice the loop upper bound for the **`FOR`** loop on line `220` is specified in terms of the variable **`N`** so if the number of elements changes, the loop code does not need to be modified. This same technique is used for the **`FOR`** loops which begin on lines `330`, `340`, and `470`.

The subroutine to populate the array **`A`** on lines `420` to `500` could be modified to read data from **`DATA`** statements, but of course in that case it would be more efficient to just sort the data outside the program and write the **`DATA`** statement with the elements in order and avoid the cost of the sort altogether.

Note that the subroutine beginning on line `510` is used to verify that the sort worked. If the list is sorted, every element $A_I \leq A_{I+1}$, for $I = 0$ to $I = N - 2$. $N - 2$ must be the upper limit since if I was $N - 1$ the program would need to compare the element stored at that index with the element stored at the next index N. There is no element at index N in the array because the array is zero-based. We know it is zero-based because of the **`OPTION BASE 0`** on line `30`.

Notice the overall structure of the program puts the main program code at the beginning, and ends it with a **`STOP`** statement on line `170`. All of the subroutines occur after that line, and the source code ends with the required **`END`** statement on line `620`. Also, because the code is non-obvious in places, many **`REM`** statements are used to help document the program's source code.

```
10 REM BUBBLE SORT DEMO PROGRAM
20 RANDOMIZE
30 OPTION BASE 0
40 DIM A(4)
50 LET N=5
60 REM    LOAD RANDOM DATA
70 GOSUB 420
80 PRINT "LIST BEFORE SORT"
90 GOSUB 180
100 REM    SORT LIST
110 GOSUB 260
120 PRINT "LIST AFTER SORT"
130 GOSUB 180
140 REM      VERIFY LIST IS SORTED
150 GOSUB 510
160 PRINT "PASS"
170 STOP
180 REM ***********************************************
190 REM      SUBROUTINE TO DISPLAY ARRAY A ELEMENTS
200 REM      N IS NUMBER OF ELEMENTS IN ARRAY A
210 REM      I IS LOOP INDEX
220 FOR I=0 TO N-1
230 PRINT "A(";I;")=";A(I)
240 NEXT I
250 RETURN
260 REM ***********************************************
270 REM      SUBROUTINE TO BUBBLE SORT ARRAY A ELEMENTS
280 REM      INTO ASCENDING ORDER
290 REM      N IS NUMBER OF ELEMENTS IN ARRAY A
300 REM      I AND J ARE LOOP INDICES
310 REM      T IS A TEMPORARY VARIABLE USED FOR SWAPPING ELEMENTS
320 REM
330 FOR I=N-1 TO 1 STEP -1
340 FOR J=0 TO I-1
350 IF A(J)<=A(J+1) THEN 390
360 LET T=A(J)
370 LET A(J)=A(J+1)
380 LET A(J+1)=T
390 NEXT J
400 NEXT I
410 RETURN
420 REM ***********************************************
430 REM      SUBROUTINE TO FILL ARRAY A WITH RANDOM
440 REM      POSITIVE INTEGER VALUES
450 REM      N IS NUMBER OF ELEMENTS IN ARRAY A
460 REM      I IS LOOP INDEX
470 FOR I=0 TO N-1
480 LET A(I)=INT(RND*100)
490 NEXT I
500 RETURN
510 REM ***********************************************
520 REM      SUBROUTINE TO VERIFY ARRAY IS SORTED
530 REM      N IS NUMBER OF ELEMENTS IN ARRAY A
540 REM      I IS LOOP INDEX
550 FOR I=0 TO N-2
560 IF A(I)>A(I+1) THEN 590
570 NEXT I
580 RETURN
590 PRINT "FAIL: ARRAY NOT SORTED CORRECTLY"
600 GOSUB 180
610 REM INTENTIONALLY FALL THROUGH TO END PROGRAM
620 END
```

Figure 14.2: Bubble sort program

Exercises

1. Some people have implemented the bubble sort algorithm in BASIC with code like this:

```
330 FOR I=1 TO N
340 FOR J=0 TO N-2
350 IF A(J)<=A(J+1) THEN 390
360 LET T=A(J)
370 LET A(J)=A(J+1)
380 LET A(J+1)=T
390 NEXT J
400 NEXT I
```

   Why is this version of the code slower in the average case than the code presented in this chapter?

2. What happens when we use the sort on lines 330 through 400 from the program shown in figure 14.2 on page 119 to sort a list that is already sorted? Improve the code so that once the list is already sorted, the sort subroutine jumps to the **RETURN** statement.

3. The bubble sort algorithm can be modified so that it is a descending sort instead of an ascending sort as shown in this chapter. Update the program from the previous question so that it performs a descending sort.

Chapter 15

Binary Search

This chapter describes an effective searching technique called *binary search* for use with data values stored contiguously in sorted, integer-indexed arrays. This search will find any value in the array, or verify no such value exists in the array. For searching a large number of data values, binary search works much more quickly than the sequential search described in chapter 12. To use this algorithm on an unsorted list stored in an integer indexed array, the array must be sorted first. One simple way to sort arrays is the bubble sort described in chapter 14. Since there is a significant cost to sorting an array, the binary search should only be used if the array will be searched many times. If you will only search an unsorted array once, a sequential search is actually faster since the runtime cost of the sort is eliminated.

Consider a sorted list of 8 elements stored in a one-dimensional, integer-indexed, zero-based array called A. This array can be represented by the following diagram:

	0	1	2	3	4	5	6	7
Array A	1	7	13	22	23	28	29	37

If we know nothing about the data except that it is sorted, when we begin the search we would naturally choose the middle element. How can this middle element be found? Create a variable to contain the lowest element index L, a variable H to contain the highest element index, and finally a variable M to contain the middle element index. The formula[1] to find the value of M from H and L is

$$M = L + \left\lfloor \frac{H - L}{2} \right\rfloor$$

When the algorithm starts, $L = 0$ and $H = 7$. The formula produces 3 for the value of M. If A_M is the value we seek, then our search has completed successfully. If A_M is not the value we seek, then since the list is sorted, it must either be greater than or less than the value we seek. If A_M is greater than the value we seek, then all values in A_X where $X \geq M$ will be greater than the value we seek, so we can

[1] Remember, $\lfloor x \rfloor$ is the math floor(x) function we first used in chapter 7.

ignore the part of the list where indices are greater than or equal to M. In other words, we can say $H = M - 1$. On the other hand, if A_M is less than the value we seek, then all values in A_X where $X \leq M$ will be less than the value we seek, so we can ignore the part of the list where indices are less than or equal to M. In other words, we can say $L = M + 1$. So in the case where A_M does not contain the element we seek, at least we can cut down our search by removing approximately half of the elements from our future searches. We can repeat this process, always looking at the middle element of the remaining list, until A_M is the value we seek, or the A_M is not the value we seek **and** the length of the list, $(H - L)$, is one.

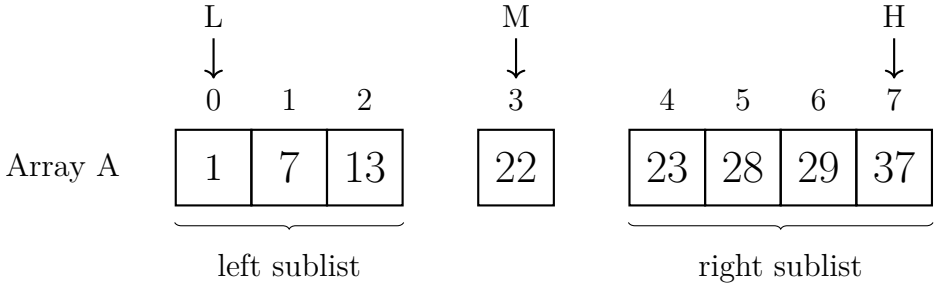
While the preceding paragraph is accurate, it may be confusing to some readers. Going through some binary searches step-by-step with diagrams will illustrate how the algorithm works.

15.1 Binary Search Example 1

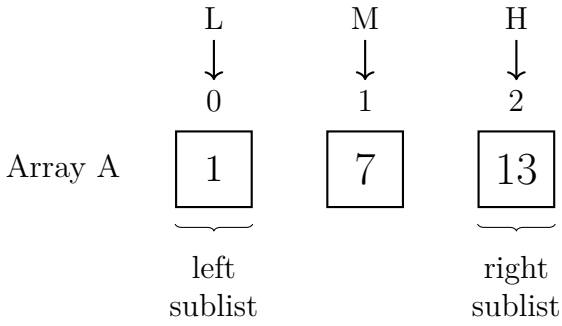
In this first example, the value we seek is 7. Initially, $H = 7$ and $L = 0$.

$$M = L + \left\lfloor \frac{H - L}{2} \right\rfloor = 0 + \left\lfloor \frac{7 - 0}{2} \right\rfloor = \lfloor 3.5 \rfloor = 3$$

The formula for M yields the value 3. The following diagram shows that the index M partitions the list into three sublists, a left sublist with all elements having values less than or equal to A_M, the list with just one element M in the middle, and a right sublist with all elements having values greater than or equal to A_M.



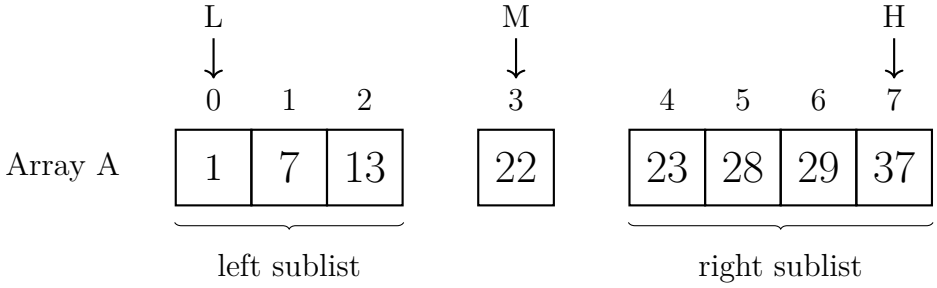
The first thing to check is whether A_M holds the value X we seek. A_3 holds the value 22, not 7, so the search is not successful yet. Since the array values are in ascending order, and we know there were no duplicates in the list, obviously *if* the value is in the array, it must be in either the left or the right sublist. When the value in A_M is compared with the value in X, we learn that 22 is greater than 7, so *if* the value is in the array, it must be in the *left* sublist. It cannot be in the right sublist since everything in that list must be greater than or equal to the value in A_M. This means that H needs to be changed to point to the last value in the left sublist, which is $M - 1$. Then we will start again using the new smaller list.



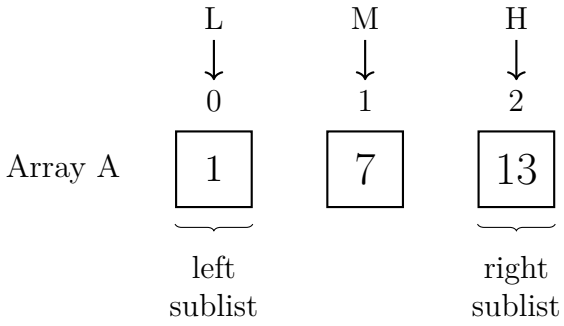
The formula for M yields the value 1. The next thing to check is whether or not A_M holds the value in X that we seek. In this case A_1 does hold the value 7, so the search has been successful.

15.2 Binary Search Example 2

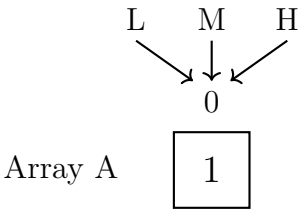
In this example, the value we seek is 3. The formula for M yields the value 3. The following diagram shows that the index M partitions the list into three sublists, a left sublist with all elements having values less than or equal to A_M, the list with just one element M in the middle, and a right sublist with all elements having values greater than or equal to A_M.



The first thing to check is whether A_M holds the value X we seek. A_3 holds the value 22, not 3, so the search is not successful yet. Since the array values are in ascending order, and we know there were no duplicates in the list, obviously *if* the value is in the array, it must be in either the left or the right sublist. When the value in A_M is compared with the value in X, we learn that 22 is greater than 3, so *if* the value is in the array, it must be in the *left* sublist. It cannot be in the right sublist since everything in that list must be greater than or equal to the value in A_M. This means that H needs to be changed to point to the last value in the left sublist, which is $M - 1$. Then we will start again using the new smaller list.



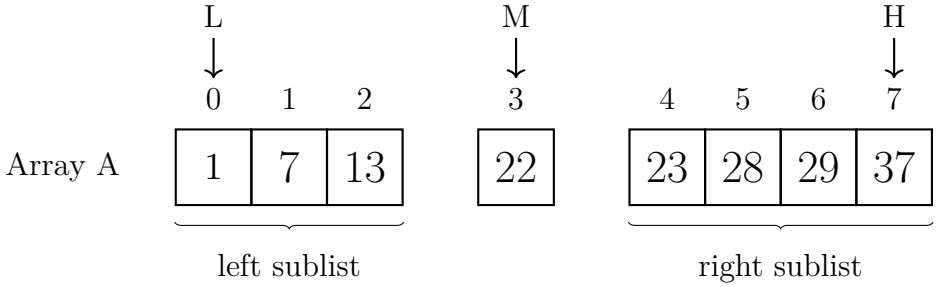
The formula for M yields the value 1. The next thing to check is whether or not A_M holds the value in X that we seek. A_1 holds the value 7, not 3, so the search is not successful yet. When the value in A_M is compared with the value in X, we learn that 7 is greater than 3, so *if* the value is in the array, it must be in the left sublist. This means that H needs to be changed to point to the last value in the left sublist, which is $M - 1$. Then we will start again using the new smaller list.



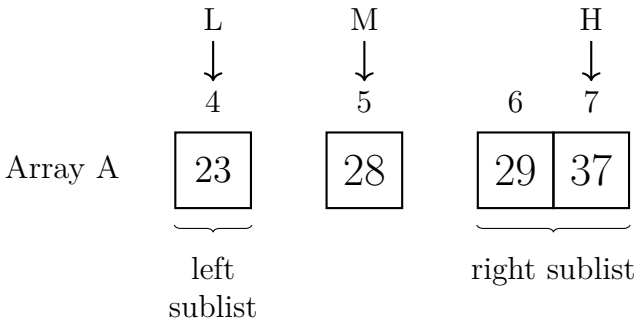
The formula for M yields the value 0. The next thing to check is whether or not A_M holds the value in X that we seek. A_0 holds the value 1, not 3, so the search is not successful yet. When the value in A_M is compared with the value in X, we learn that 1 is less than 3, so *if* the value is in the array, it must be greater than or equal to the value in A_M. This means that L needs to be changed to point to the first value in the right sublist, which is $M + 1$. The picture makes it obvious that in this case that both the left and right sublists are empty. At this point $L = 1$ and $H = 0$. When L is greater H, that means the list has no elements. Now we know that 3 is not in the list, so the search was not successful. The algorithm worked, but the value sought was not in the array, so the search was not successful.

15.3 Binary Search Example 3

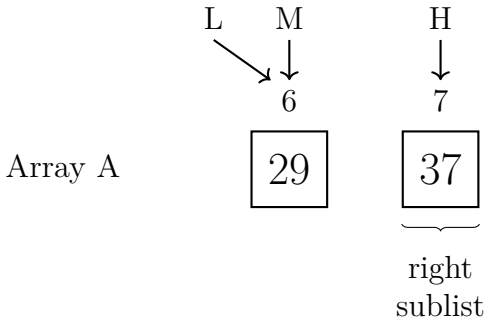
In this example, the value we seek is 37. The formula for M yields the value 3. The following diagram shows that the index M partitions the list into three sublists, a left sublist with all elements having values less than or equal to A_M, the list with just one element M in the middle, and a right sublist with all elements having values greater than or equal to A_M.



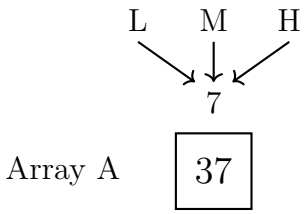
The first thing to check is whether A_M holds the value X we seek. A_3 holds the value 22, not 37, so the search is not successful yet. Since the array values are in ascending order, and we know there were no duplicates in the list, obviously *if* the value is in the array, it must be in either the left or the right sublist. When the value in A_M is compared with the value in X, we learn that 22 is less than 37, so *if* the value is in the array, it must be in the *right* sublist. It cannot be in the left sublist since everything in that list must be less than or equal to the value in A_M. This means that L needs to be changed to point to the first value in the right sublist, which is $M + 1$. Then we will start again using the new smaller list.



The formula for M yields the value 5. The next thing to check is whether or not A_M holds the value in X that we seek. A_5 holds the value 28, not 37, so the search is not successful yet. Since the array values are in ascending order, and we know there were no duplicates in the list, obviously *if* the value is in the array, it must be in either the left or the right sublist. When the value in A_M is compared with the value in X, we learn that 28 is less than 37, so *if* the value is in the array, it must be in the *right* sublist. It cannot be in the left sublist since everything in that list must be less than or equal to the value in A_M. This means that L needs to be changed to point to the first value in the right sublist, which is $M + 1$. Then we will start again using the new smaller list.



The formula for M yields the value 6. The next thing to check is whether or not A_M holds the value in X that we seek. A_6 holds the value 29, not 37, so the search is not successful yet. Since the array values are in ascending order, and we know there were no duplicates in the list, obviously *if* the value is in the array, it must be in either the left or the right sublist. When the value in A_M is compared with the value in X, we learn that 29 is less than 37, so *if* the value is in the array, it must be in the *right* sublist. It cannot be in the left sublist since everything in that list must be less than or equal to the value in A_M. This means that L needs to be changed to point to the first value in the right sublist, which is $M + 1$. Then we will start again using the new smaller list.



The formula for M yields the value 7. The next thing to check is whether or not A_M holds the value in X that we seek. In this case A_7 does hold the value 37, so the search has been successful.

15.4 Binary Search in Detail

A flowchart showing the logic of an iterative implementation of the binary search algorithm is shown in figure 15.1 on page 127. The flowchart clearly shows the looping nature of the algorithm. Each iteration of the loop body will process the list of values stored in the array once. For the common case when $A_M \neq X$, three comparisons must be made. The first is to ensure the list has at least one element, the second is to see if $A_M = X$, and the third is to determine which sublist to process on the next iteration of the loop. In the successful case, only the first two comparisons are made. In the failure case, the first comparison will be true and the algorithm will terminate.

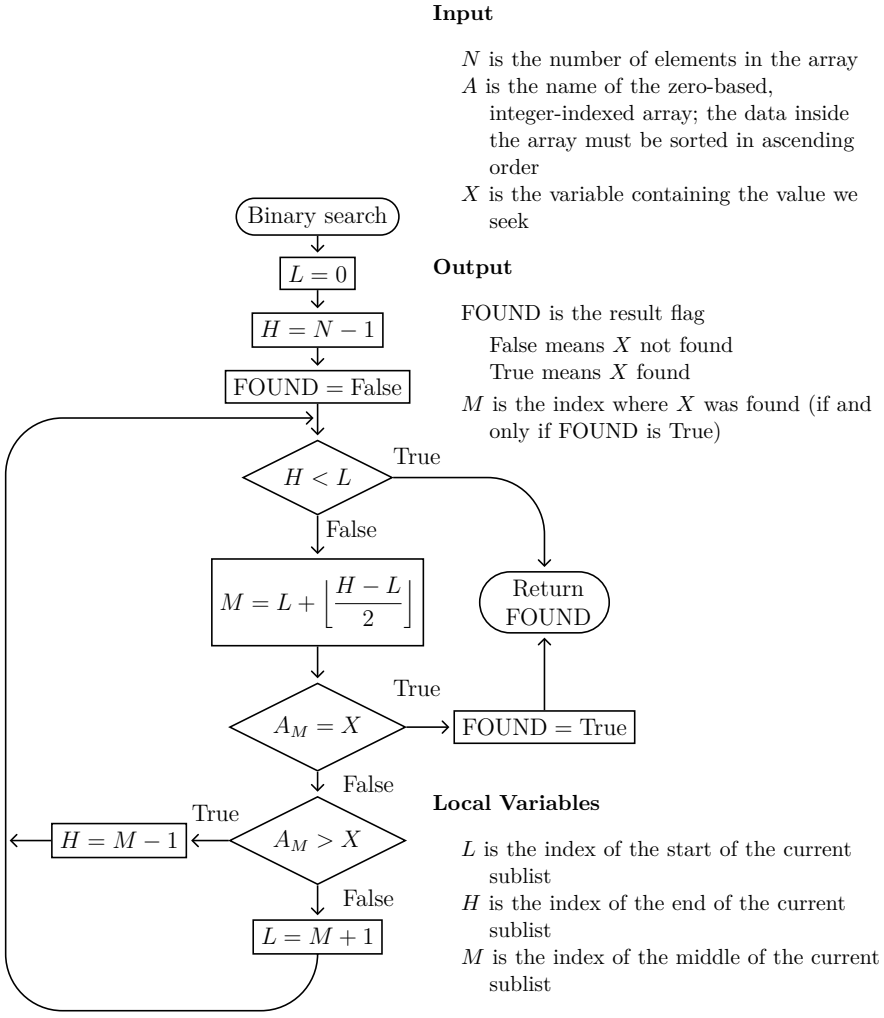


Figure 15.1: Flowchart for iterative binary search of array of N elements

15.5 Performance of Binary Search

The version of the binary search used in this chapter requires a maximum of three comparisons for an iteration of the loop. However, sequential search only requires two comparisons on each inner loop iteration. For short lists, sequential search will be faster than binary search. For larger lists, though, binary search works much better. Consider the case where the number of elements is one million. So how many iterations of the loop will we need? As shown in figure 15.1, only twenty iterations is enough to find any value in the list. Even with three comparisons in every case, that is only sixty comparisons in the worst case, compared to at least one million comparisons in the worst case for sequential search.

Loop Iteration	Remaining List Size (Number of Elements)
1	1000000
2	500000
3	250000
4	125000
5	62500
6	31250
7	15625
8	7812
9	3906
10	1953
11	976
12	488
13	244
14	122
15	61
16	30
17	15
18	7
19	3
20	1

Table 15.1: How many iterations are required for a large binary search?

Binary search is an $O(\log_2 n)$ algorithm for the number of loop iterations on average and in the worst case. A graph comparing various curves including $y = x$, which corresponds to $O(n)$ for sequential search, to $y = \log_2 x$, which corresponds to $O(\log_2 n)$ for binary search, is shown in figure 15.2 on page 129.

Once n is greater than about 10, these graphs show that the behavior of the different big-O curves is dramatically different. The chart in table 15.2 shows what happens as the input size n increases for several of the curves.

If n is doubled then it takes		Used by
twice as many operations	$O(n)$	sequential search
four times as many operations	$O(n^2)$	bubble sort
one more operation	$O(\log_2 n)$	binary search
no additional operations	$O(1)$	access a global variable

Table 15.2: $O(n)$ table

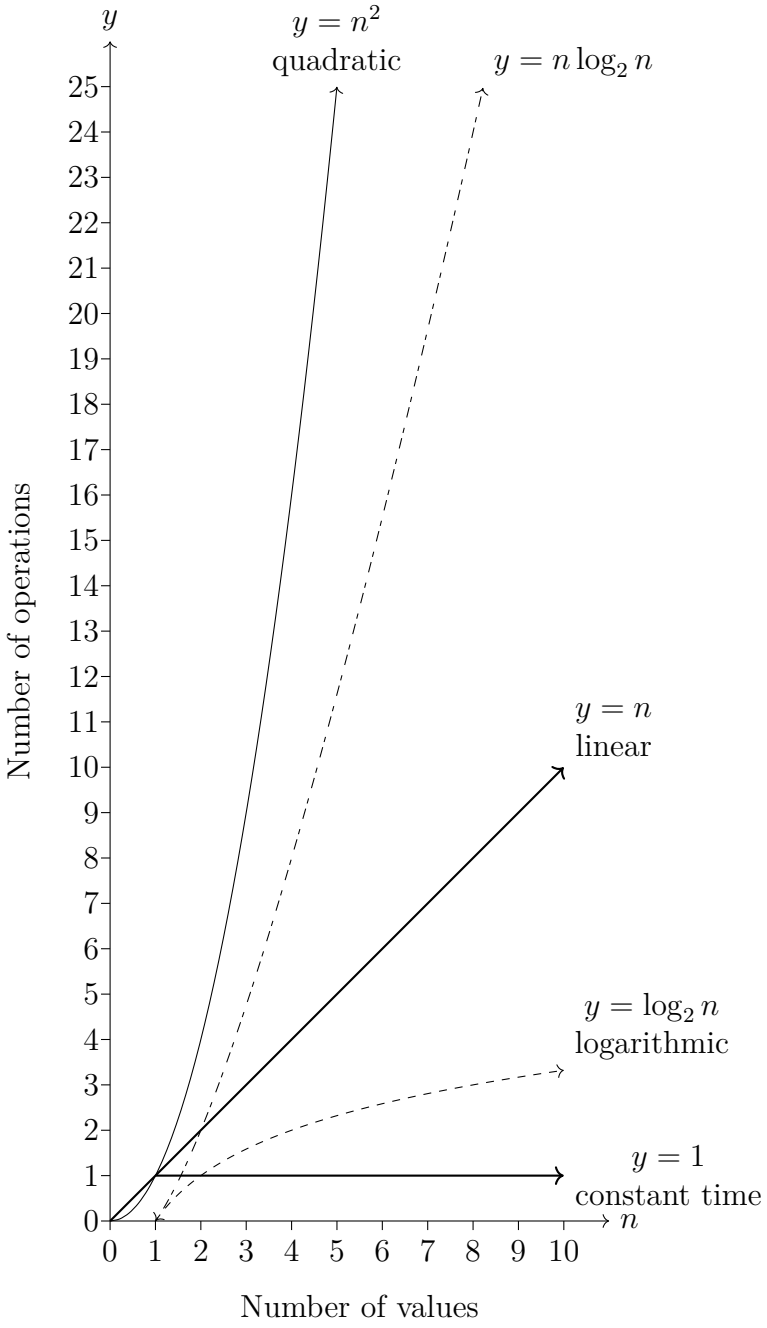


Figure 15.2: $O(n)$ Graphs

15.6 Implementation in ECMA-55 Minimal BASIC

An ECMA-55 Minimal BASIC implementation of iterative binary search of a zero-based, integer-indexed array of numbers is shown in figure 15.3 on page 131. This implementation reads unsorted numeric data into an array in the subroutine on line 300, and lines 320 through 350 correspond to the flowchart in figure 13.2 on page 109. Once the data is loaded, the program then uses a bubble sort in the subroutine on line 400 to put the values into ascending order and lines 410 through 490 correspond to the flowchart in figure 14.1 on page 117. The actual binary search code is in the subroutine that starts on line 500 and corresponds to the flowchart in figure 15.1 on page 127. The array variable `A` defined on line 30 can hold up to 100 values, leaving room for adding more values later. To use more values requires modifying the `DATA` statement on line 40. The first value in that statement is the number of elements to load, in this case 8. Following that are the values. If you wanted to load a longer list, you would change that 8 to the number of elements you want to load, and you would add more `DATA` statements with the additional numeric values.

If you consider binary search for a while, you will realize it is related to the high-low game from chapter 7. The high-low game stated that player one must choose a random integer value with a range. Player two knows the range. If player two chooses the midpoint value and is not correct, at least the player knows from the "high" or "low" response that $\frac{1}{2}$ of the list absolutely does NOT contain the value, and player two can change the range to the new smaller range. This is $O(\log_2 n)$ and that is how we can know that for the range 1 to 100, the maximum number of guesses will be $\lceil \log_2 100 \rceil$, which is approximately, $\lceil 6.65 \rceil$, which is 7.

```
10 REM           BINARY SEARCH
20 OPTION BASE 0
30 DIM A(99)
40 DATA 8,37,29,28,23,22,13,7,1
50 GOSUB 300
60 GOSUB 400
70 PRINT "WHAT VALUE DO YOU SEEK";
80 INPUT X
90 GOSUB 500
100 IF F=0 THEN 130
110 PRINT A(M);"FOUND AT INDEX";M
120 GOTO 140
130 PRINT X;"NOT FOUND"
140 STOP
300 REM          LOAD ARRAY
310 READ N
320 FOR I=0 TO N-1
330 READ A(I)
340 NEXT I
350 RETURN
400 REM BUBBLE SORT
410 FOR I=N-1 TO 1 STEP -1
420 FOR J=0 TO I-1
430 IF A(J)<=A(J+1) THEN 470
440 LET T=A(J)
450 LET A(J)=A(J+1)
460 LET A(J+1)=T
470 NEXT J
480 NEXT I
490 RETURN
500 REM          BINARY SEARCH
510 REM N IS NUMBER OF ELEMENTS
520 REM A IS ARRAY OF ELEMENTS
530 REM X IS VALUE WE SEEK
540 REM F IS FLAG, 0 IS NOT FOUND, 1 IS FOUND
550 REM L IS LOWEST INDEX
560 REM H IS HIGHEST INDEX
570 REM M IS MIDDLE INDEX
580 LET L=0
590 LET H=N-1
600 LET F=0
610 IF H<L THEN 730
620 LET M=L+INT((H-L)/2)
630 IF A(M)=X THEN 710
640 IF A(M)>X THEN 680
650 REM A(M) IS TOO SMALL
660 LET L=M+1
670 GOTO 610
680 REM A(M) IS TOO BIG
690 LET H=M-1
700 GOTO 610
710 REM FOUND IT
720 LET F=1
730 RETURN
740 END
```

Figure 15.3: Binary search program

Exercises

1. What happens when we search for an item with a value less than all values in the list? For example, using the list shown on page 121, what happens if we search for the value -3?

2. What happens when the array to be searched contains some duplicate values? Update the flowchart shown in figure 15.1 on page 127 to report the position of the **first** value in the array that matches value X being sought. One way to do this is to do the normal binary search, then if the value was found at A_M, keep reducing M until M is negative or the value in A_M differs from X. At that point A_{M+1} is the first occurrence of the value X in the array A.

3. Modify the program shown in figure 15.3 on page 131 so that implements the search from the flowchart created in the answer to the previous question.

4. Modify the flowchart shown in figure 15.1 on page 127 so that it works with an array that is sorted in **descending** order.

5. Modify the program shown in figure 15.3 on page 131 so that implements the search from the flowchart created in the answer to the previous question.

6. Update the program shown in figure 15.3 on page 131 to use one-based arrays.

Chapter 16

Two-Dimensional Arrays

This chapter introduces two-dimensional, integer-indexed arrays of numbers. A one-dimensional array is much like a vector, but a two-dimensional array is like a *matrix*. In Minimal BASIC, all elements of an array must be numeric, and an array's number of dimensions is the number of subscripts used to address an element. A one-dimensional array element is accessed using a single subscript, but a two-dimensional array element is accessed using two subscripts. Minimal BASIC, like C, C++, and Java, uses a row-major layout for matrices. This means that the first subscript specifies which row to access, and the second subscript specifies which column to access. Like one-dimensional arrays, two-dimensional arrays are declared using the `DIM` statement. For example, `DIM Z(5,3)` would create the array Z where the maximum row index is 5 and the maximum column index is 3. For zero-based arrays, that creates a matrix with 6 rows and 4 columns. For one-based arrays, that creates a matrix with 5 rows and 3 columns. It is always best to include an `OPTION BASE` statement before any `DIM` statements so that there is no doubt about whether you want zero or one as the minimum index value. The logical layout of array Z when using zero-based arrays is shown in figure 16.1. The logical layout of array Z when using one-based arrays is shown in figure 16.2 on page 134.

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	6	7	8	9	10	11
2	12	13	14	15	16	17
3	18	19	20	21	22	23

Figure 16.1: Layout of a zero-based 6x4 matrix

An example program will help you understand how to use two-dimensional arrays. The example program shown in figure 16.9 on page 141 and figure 16.10 on page 142

	1	2	3	4	5
1	0	1	2	3	4
2	5	6	7	8	9
3	10	11	12	13	14

Figure 16.2: Layout of a one-based 5x3 matrix

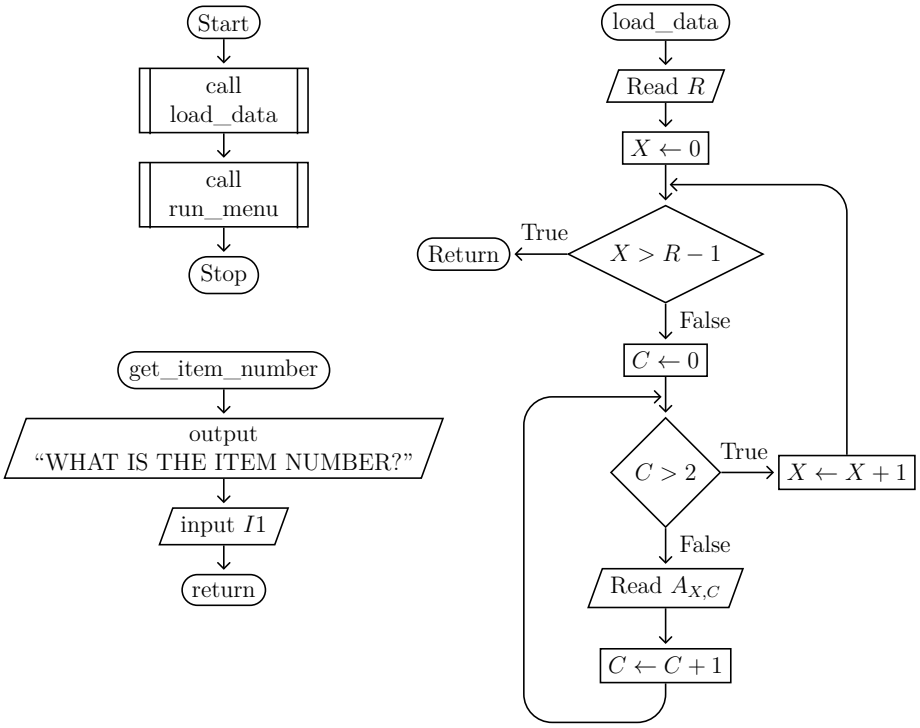


Figure 16.3: main, load_data, and get_item_number

is a program that allows querying an inventory of items. The data available includes the unit price and the quantity. Since Minimal BASIC does not support string arrays, we must use a numeric product identifier for each product. Each row of array `A` contains information about one product. The first column contains the product identifier number, the second column contains the price, and the third column contains the quantity. The rows are unsorted, so sequential search is used when looking up items by their numeric product identifier. Since the maximum number of rows is only 10, the $O(n)$ performance of sequential search will not be a problem to worry about for this program.

A *record* is a compound type composed of multiple units of data. In this chapter we use one row of the matrix to represent one record. Each unit of data in a record

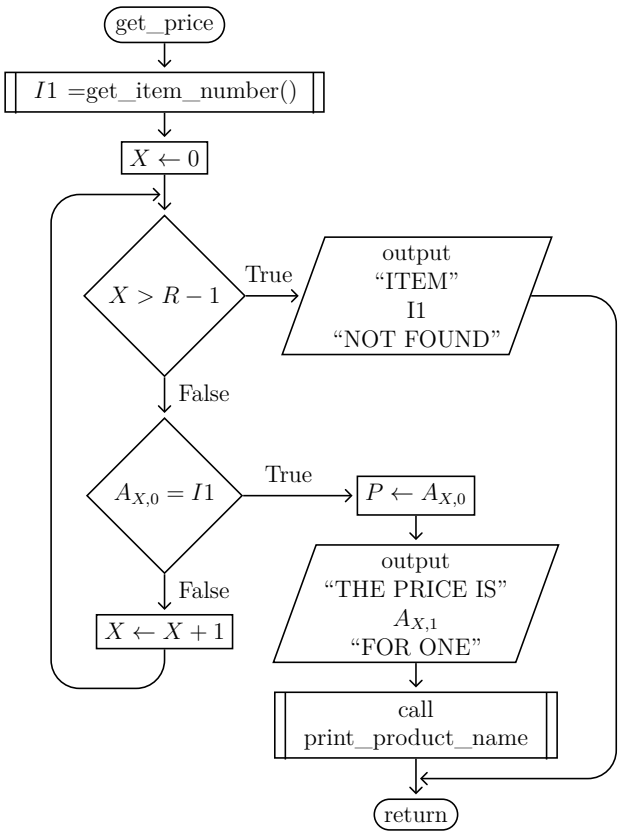


Figure 16.4: get_price subroutine

is called a *field*. When using a matrix row for a record, each field is stored in a different column. So the first index of each matrix reference identifies the record number, and the second index identifies the field. This allows using a matrix to hold an array of records, even though ECMA-55 Minimal BASIC has no support for records included in the language.

The program is structured into several subroutines to make it easier to read and modify. Of course it is not possible to query data until it is loaded into the array, so the first subroutine called is the one to load the data on line 2000. The number of rows is first read into the variable **`R`**, then the data for **`R`** rows, each with 3 columns, is read from **`DATA`** statements. After the data is loaded, it is time to ask the user what information they want to see. The flowchart for *load_data* is shown in figure 16.3 on page 134.

The subroutine for the main menu starts on line 1000. It displays the menu on lines 1010 through 1050, then gets the user choice on lines 1060 through 1070. That choice is checked to ensure it is in the range 1 through 4 inclusive on lines 1080 through 1110. If it invalid, a message is displayed and the subroutine then lets the user try again. If it is valid, then the multi-way branch on line 1120 will

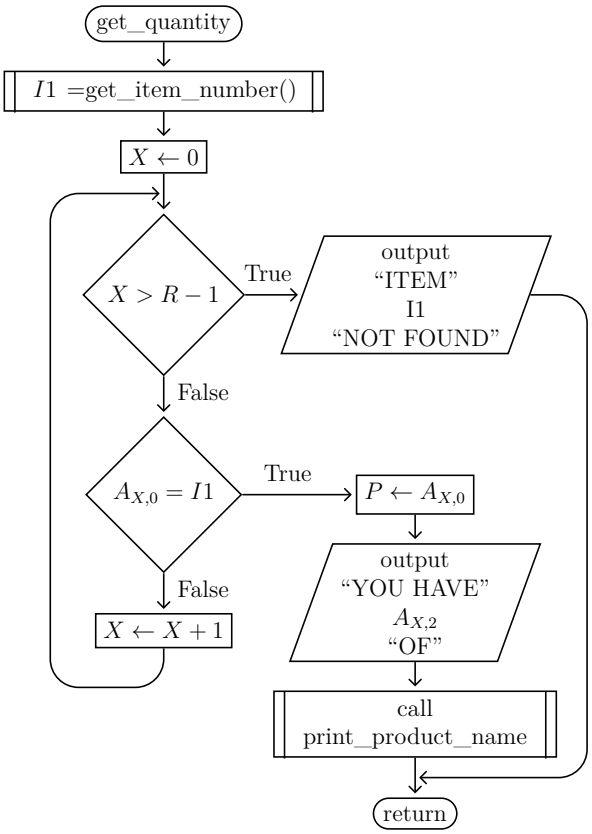


Figure 16.5: get_quantity subroutine

jump to appropriate lines to run the subroutines that implement the first three choices, or to return from the subroutine if the user chooses to exit the program. The flowchart for *run_menu* is shown in figure 16.6 on page 137.

The first menu choice allows the user to ask for the price of an item. The subroutine to do this begins on line 4000. The subroutine first calls another subroutine on line 8000 to get the numeric product identifier, called the item number in this program, from the user. The flowchart for *get_item_number* is shown in figure 16.3 on page 134. That subroutine puts the product identifier into the variable `I1`. Then it returns and the program performs a sequential search through array `A` examining the first column of each of the `R` rows to see if the product id in the first column matches the product id stored in `I1`. This check is on line 6040. If the product id is found in the array, then line 6050 stores that ID in the variable `P` which is the parameter for the subroutine that prints the item name on line 3000. It then prints the price on line 6060 and ends the print statement with a semicolon to prevent a newline from being sent. The subroutine on line 3000 is then called which will print the product name that corresponds to the product identifier stored in the variable `P` using a long series of `IF` statements. An `ON..GOTO` statement cannot be used since the values of P are not easily convertible to the sequence

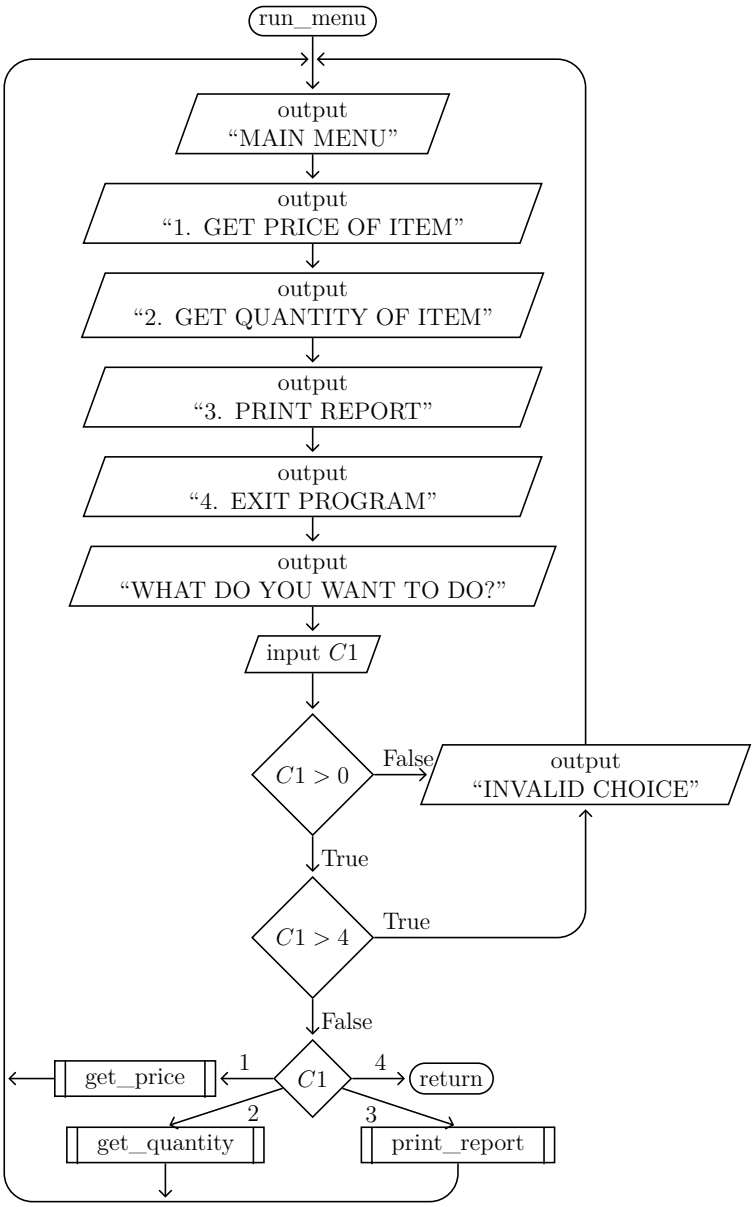


Figure 16.6: run_menu subroutine

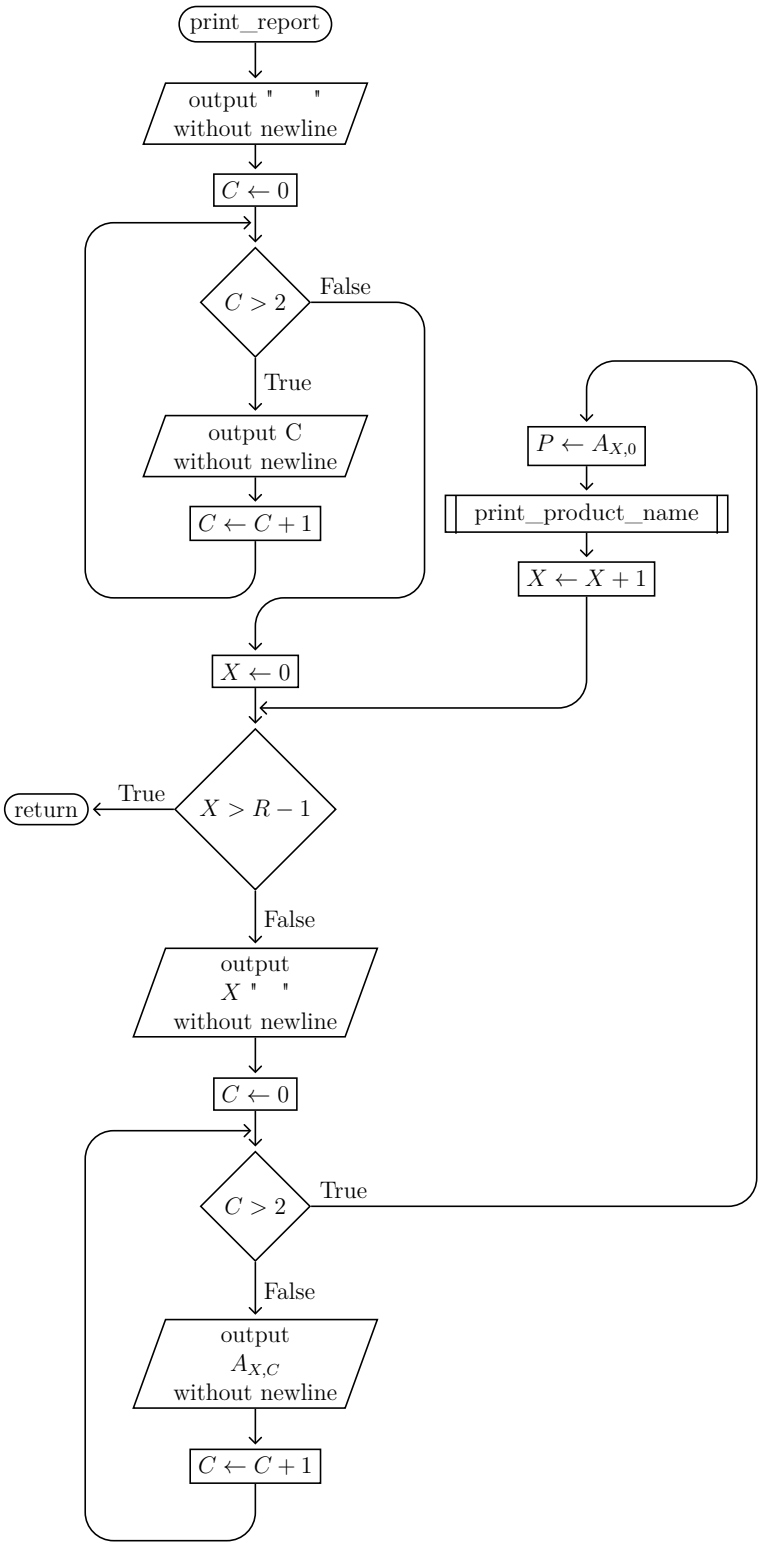


Figure 16.7: print_report subroutine

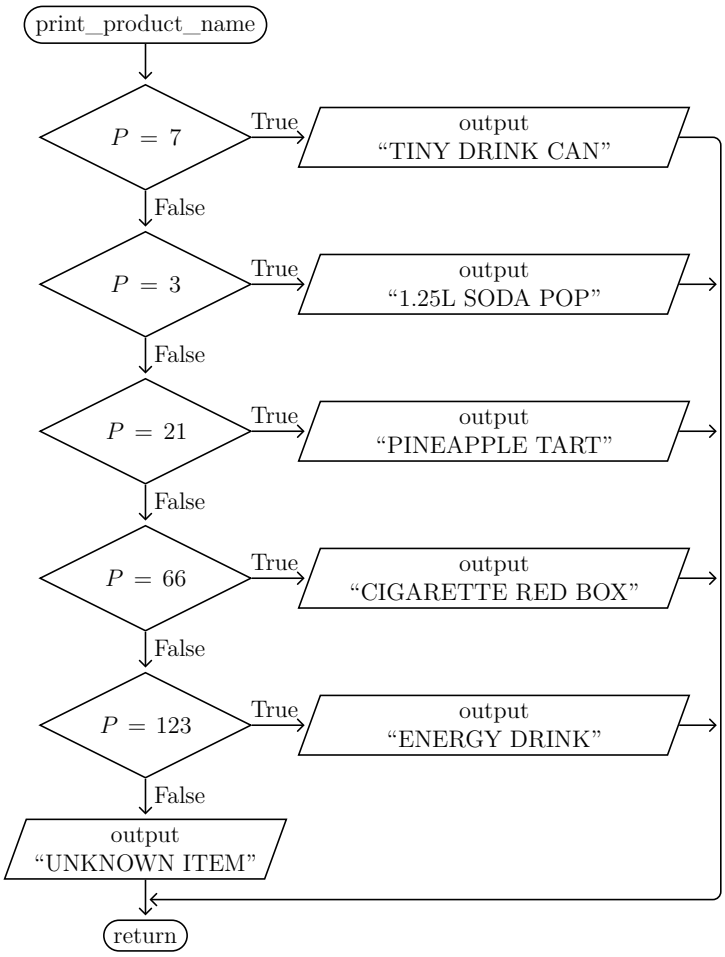


Figure 16.8: print_product_name subroutine

$1, 2, 3, \ldots$ required to use that statement. The **`PRINT`** statements in the subroutine that starts on line 3000 do not end with a semicolon or a comma, so they will send the newline and finish the output line started in the subroutine on line 4000. The flowchart for the *get_price* subroutine is shown in figure 16.4 on page 135. It calls the *get_item_number* subroutine which has a flowchart shown in figure 16.3 on page 134, and the *print_product_name* subroutine which has a flowchart shown in figure 16.8.

The second menu choice allows the user to ask for the quantity of an item. The subroutine to do this begins on line 6000 is essentially identical to the subroutine to show the price of an item. The only difference is that column 2 is printed instead of column 1, and the text message on line 6060 is a little bit different. It also uses the subroutines at 8000 and 3000 in the same way as the subroutine at 4000 did. The flowchart for *get_quantity* is shown in figure 16.5 on page 136. Like *get_price*, *get_quantity* calls the *get_item_number* which has a flowchart shown

in figure 16.3 on page 134, *get_item_number* subroutine which has a flowchart shown in figure 16.3 on page 134, and the *print_product_name* subroutine which has a flowchart shown in figure 16.8 on page 139.

The third menu choice allows the user to just display the entire table of data. The subroutine to do this begins on line 2500. The column numbers are printed first on lines 2510 through line 2550. Then the rows are printed one-by-one on lines 2560 through line 2630. Like the first and second menu choices, the product name is displayed using the subroutine at 3000. The subroutine for *print_report* is shown in figure 16.7 on page 138, and the subroutine to print the product name is the same one as used in *get_price* and *get_quantity* and is shown in figure 16.8 on page 139.

```
10 REM           TWO-DIMENSIONAL READ-ONLY INVENTORY DEMO PROGRAM
20 REM
30 REM  A    THE ARRAY OF DATA
40 REM  R    THE NUMBER OF ROWS USED IN A
50 REM  C1   MENU CHOICE ENTERED BY USER
60 REM  X    INDEX TO MOVE THROUGH ROWS IN A
70 REM  C    INDEX TO MOVE THROUGH COLUMNS IN A
80 REM  P    PRODUCT ID PARAMETER USED FOR PRINTING PRODUCT NAME
90 REM  I1   PRODUCT ID PARAMETER USED FOR SEARCHING FOR PRODUCT
100 REM
110 REM *** MAIN ***
120 OPTION BASE 0
130 DIM A(9,2)
140 REM    LOAD THE DATA
150 GOSUB 2000
160 REM    RUN MENU UNTIL THEY QUIT
170 GOSUB 1000
180 STOP
190 REM
1000 REM *** MAIN MENU SUBROUTINE ***
1010 PRINT "         MAIN MENU"
1020 PRINT "1.  GET PRICE OF ITEM"
1030 PRINT "2.  GET QUANTITY OF ITEM"
1040 PRINT "3.  PRINT REPORT"
1050 PRINT "4.  EXIT PROGRAM"
1060 PRINT "WHAT DO YOU WANT TO DO";
1070 INPUT C1
1080 IF C1>0 THEN 1110
1090 PRINT "INVALID CHOICE"
1100 GOTO 1010
1110 IF C1>4 THEN 1090
1120 ON C1 GOTO 1130,1160,1210,1190
1130 REM GET PRICE
1140 GOSUB 4000
1150 GOTO 1010
1160 REM GET QUANTITY
1170 GOSUB 6000
1180 GOTO 1010
1190 REM EXIT PROGRAM
1200 RETURN
1210 REM PRINT REPORT
1220 GOSUB 2500
1230 GOTO 1010
1240 REM
2000 REM *** LOAD DATA SUBROUTINE ***
2010 READ R
2020 FOR X=0 TO R-1
2030 FOR C=0 TO 2
2040 READ A(X,C)
2050 NEXT C
2060 NEXT X
2070 RETURN
2080 REM
2500 REM *** DISPLAY DATA SUBROUTINE ***
2510 PRINT "     ";
2520 FOR C=0 TO 2
2530 PRINT C,
2540 NEXT C
2550 PRINT
2560 FOR X=0 TO R-1
2570 PRINT X;": ";
2580 FOR C=0 TO 2
2590 PRINT A(X,C),
2600 NEXT C
2610 LET P=A(X,0)
2620 GOSUB 3000
2630 NEXT X
2640 RETURN
2650 REM
```

Figure 16.9: Logical Array of Records program (1 of 2)

```
3000 REM *** PRINT PRODUCT NAME FOR PRODUCT P SUBROUTINE ***
3010 IF P<>7 THEN 3040
3020 PRINT "TINY DRINK CAN"
3030 RETURN
3040 IF P<>3 THEN 3070
3050 PRINT "1.25L SODA POP"
3060 RETURN
3070 IF P<>21 THEN 3100
3080 PRINT "PINEAPPLE TART"
3090 RETURN
3100 IF P<>66 THEN 3130
3110 PRINT "CIGARETTE RED BOX"
3120 RETURN
3130 IF P<>123 THEN 3160
3140 PRINT "ENERGY DRINK"
3150 RETURN
3160 PRINT "UNKNOWN ITEM"
3170 RETURN
3180 REM
4000 REM *** GET PRICE OF ITEM SUBROUTINE ***
4010 GOSUB 8000
4020 REM SEQUENTIAL SEARCH FOR ITEM I1
4030 FOR X=0 TO R-1
4040 IF A(X,0)<>I1 THEN 4090
4050 LET P=A(X,0)
4060 PRINT "THE PRICE IS";A(X,1);"FOR ONE ";
4070 GOSUB 3000
4080 GOTO 4110
4090 NEXT X
4100 PRINT "ITEM";I1;"NOT FOUND"
4110 RETURN
4120 REM
6000 REM *** GET QUANTITY OF ITEM SUBROUTINE ***
6010 GOSUB 8000
6020 REM SEQUENTIAL SEARCH FOR ITEM I1
6030 FOR X=0 TO R-1
6040 IF A(X,0)<>I1 THEN 6090
6050 LET P=A(X,0)
6060 PRINT "YOU HAVE";A(X,2);"OF ";
6070 GOSUB 3000
6080 GOTO 6110
6090 NEXT X
6100 PRINT "ITEM";I1;"NOT FOUND"
6110 RETURN
6120 REM
8000 REM *** GET ITEM NUMBER FROM USER SUBROUTINE ***
8010 PRINT "WHAT IS THE ITEM NUMBER";
8020 INPUT I1
8030 RETURN
8040 REM
9000 REM *** DATA ***
9010 REM NUMBER OF ROWS IN TABLE
9020 DATA 5
9030 REM ROW DATA (IN GROUPS OF 3 FIELDS)
9040 DATA 7,10,3,3,24,1,21,20,10,66,94,35
9050 DATA 123,10,1001
9999 END
```

Figure 16.10: Logical Array of Records program (2 of 2)

Exercises

1. Add two new items to the program shown in figure 16.9 on page 141 and figure 16.10 on page 142 and then verify the program still works. The first item to add is bologna, which has a unit price of 34 baht and a quantity of 10. The second item to add is chips with a unit price of 20 baht and a quantity of 2.

2. Convert the program shown in figure 16.9 on page 141 and figure 16.10 on page 142 to use one-based array indexing.

3. Modify the program shown in figure 16.9 on page 141 and figure 16.10 on page 142 to have an additional column for the unit cost of each item.

4. Modify the program shown in figure 16.9 on page 141 and figure 16.10 on page 142 to sort the array by product ID after loading it, and to use binary search for lookups.

Chapter 17

User-defined Functions

The ECMA-55 Minimal BASIC language has limited support for simple user-defined functions. These functions can either have no parameter or exactly one scalar numeric parameter, and they must fit on a single line. The function can return a constant value or the value of any arithmetic expression that does not directly or indirectly cause the function to call itself. An example program will help explain the syntax and semantics. The parameter, if one exists, is local to the function, but all other variables in any expression used by a user-defined function are globals. User-defined functions are declared using the **DEF** keyword. The program shown in figure 17.1 has two user-defined functions, **FNP** and **FNY**.

```
10 DEF FNP=3.1415926535
20 DEF FNY(X)=A*X^2+B*X+C
30 INPUT A,B,C
35 PRINT SIN(FNP/6),COS(FNP/6)
40 PRINT 3,FNY(3)
50 LET A=FNP/6
60 LET X=9
65 LET Z=3
70 PRINT SIN(A),COS(A)
80 PRINT Z,FNY(Z)
90 PRINT A,X
100 END
```

Figure 17.1: Example Using User-defined Functions

The **FNP** function defined on line 10 takes no argument and returns the constant value for π. This is the only way to have something like a named constant value in ECMA-55 Minimal BASIC. In the example program, the **FNP** function is called on lines 35 and 50 to help generate the angle $\frac{\pi}{6}$ in radians. This is the typical way that user-defined functions with no parameter are used. This has the advantage of simulating a read-only constant, and it also means that if you have an error in your constant, to fix it you only have to fix it once in the function definition instead of many times where it is used in the program. The angle is then used as an argument to the built-in **SIN** and **COS** functions which correspond to the well-known mathematical sine and cosine functions respectively. Finally, for a

constant value longer than 3 digits, this saves typing if you use the constant many times in the program.

The **FNY** function defined on line 20 takes one argument X and returns the value of the expression $A \times X^2 + B \times X + C$, where the variables A, B, and C are global variables and X is local to the function.[1] This kind of user-defined function is often used when you want to do something with an expression so that the subroutines that do the main work can be written to call the user-defined function instead of hard-coding the expression. Then it is easy for users to just change the **DEF** expression to change the equation to be used.

```
? 1,2,3
 .5             .866025
 3              18
 .5             .866025
 3              13.7124
 .523599        9
```

Figure 17.2: User-defined Functions Program Output

The output of 13.7124 might surprise you, but actually it demonstrates that the value A used in **FNY** is a global variable, so the assignment on line 50 changed A from 1 to $\frac{\pi}{6}$ (about 0.5235988), which in turn means that the return value of the **FNY** function will change (for any X not zero).

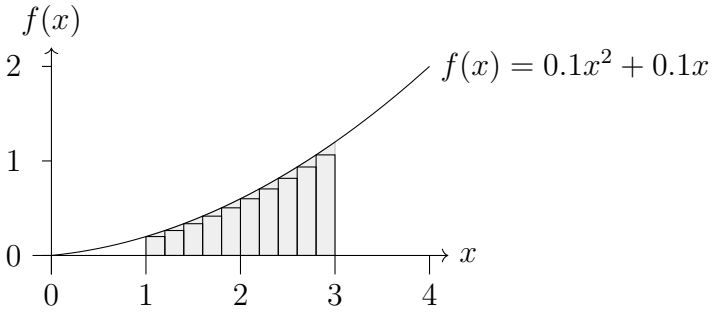
However, the assignment on line 60 changed the global X, but the call to **FNY** which has an X parameter variable did not alter the value of the global X, which remains 9 when printed on line 90, even though it was clearly 3 when the **FNY** function was evaluated on line 80.

Note well that the ECMA-55 Minimal BASIC standard specifies in section 10.4 that a function can only be defined once in a program. In practice, **DEF** statements must only occur at the beginning of the program. A **DEF** statement must never occur within the body of a **FOR** loop or within a subroutine. The control flow through the program must carefully ensure that a **DEF** is never executed more than once.

[1] This case (a parameter to a user-defined function) is the only exception to the rule that all variables in ECMA-55 Minimal BASIC are global variables.

Exercises

1. Write a flowchart to use the left Riemann sum to calculate $\int_a^b f(x)dx$, where $f(x) = 0.1x^2 + 0.1x$, $a = 1$, and $b = 3$.



   The gray area under the curve is the true area under the curve, but the sum of the area of the rectangles approximates that value. In the picture shown, the number of rectangles n is 10. As the number of rectangles increases by choosing a larger value of n the accuracy will increase. The Left Riemann sum for a function $f(x)$ is calculated using the start of the interval a, the end of the interval b, and the number rectangles n as follows:

$$w = \frac{(b-a)}{n} \qquad \sum_{i=0}^{n-1} \overbrace{\underbrace{f(a+i*w)}_{\text{left side height}} * \underbrace{w}_{\text{width}}}^{\text{Area of one rectangle}}$$

   In theory, the accuracy will increase as n increases, but limited machine precision for the floating point math eventually causes problems when n gets large enough. This technique is one form of *numeric integration*. For a polynomial it is possible to compute the exact answer to the definite integral. However, for more complicated functions numeric integration is one way to calculate the definite integral you can use even if you cannot find the anti-derivative of the function $f(x)$.

2. Write a program, using the previous question's flowchart to create a subroutine for calculating the area under the curve for $y = f(x)$, using a user-defined function for f(x).

Chapter 18

Singly Linked Lists

Often when programming you will need to manipulate lists of data. Usually the data has some unique identifier like part number, student id number, etc., and some other data that may not be unique. The unique identifier serves as a key value to access the rest of the data associated with that key. The set of all data and a key together is traditionally stored in a record. Using data as lists of records is a common programming task that every computer programmer must know. One approach to storing lists using records was covered in chapter 16. Another method for managing lists of records is called the singly linked list, and that is the method that will be described in this chapter.

When storing a list of records as a singly linked list, each record is called a *node*, and that node has one extra field added called the *next pointer*. One special variable is called the *head pointer* and it points to the first node in the list. Each node then points to the next node in the list. The final node uses a special next value called the *null pointer* to indicate that there is no next node. The diagram in figure 18.1 shows conceptually what a linked list looks like.

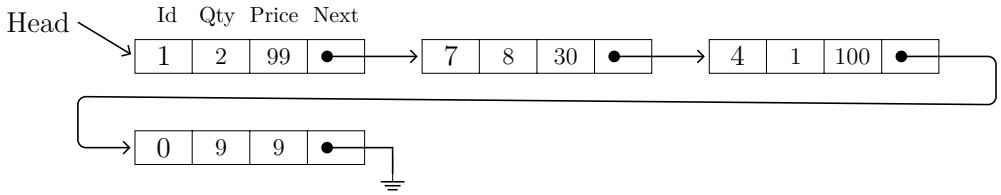


Figure 18.1: Singly Linked List

In figure 18.1 there are four nodes in the linked list, with key values 1, 4, 7, and 0. Each node contains a record with that key value in the Id field, and two other fields of data for the quantity and unit price. The Head variable is used to point the first node in the list. In many programming languages there is a special type for records, and another special type for pointers. In ECMA-55 Minimal BASIC there are no pointers or records. However, as long as all the data is numeric, it is still possible to program using linked lists using low-level programming techniques.

18.1 The Dynamic Memory Concept

When using linked lists, nodes must be allocated and deallocated from a memory area which is traditionally called a *heap*.[1] In a language with full dynamic memory support, it is possible to allocate and deallocate different sized areas of memory. Each of these areas of memory is called a *block* of memory. The runtime support code that provides functions to allocate and deallocate memory is called the dynamic memory manager.

When memory is allocated with an allocator function, the size desired is sent in as a parameter, and a pointer to the block is returned. So what exactly is a pointer? A *pointer* is just a variable which holds a number of the byte location in memory where a block is located. A pointer is normally stored in a special variable called a pointer variable. In chapter 4 you learned that a variable is just a name for a location of data in memory. A pointer variable is a name for a location in memory of where *another* location in memory is stored. A pointer is an indirect reference to a memory location. In other words, a pointer variable is a variable that can point to another location in memory. A normal variable such as a scalar variable or an array variable always points to the same address, or location, in memory for the entire duration of the program. A pointer variable, however, can point to different locations in memory at different times. A programmer can change what memory the pointer variable will access with a simple assignment statement.

You can see the difference between a pointer variable and a scalar variable in figure 18.2 on page 151. In the figure, $PointerVar$ has the value 8184, so it indirectly references the value 999. If you updated $PointerVar$ to have the value 8200, it would instead indirectly access the value 13. In other words, the $PointerVar$ variable does not hold a value directly like $ScalarVar$, but instead holds the address of another memory location.

Usually after you use an allocated memory block, you will be done with it and will recycle it. This is achieved by moving the block to a special linked list called the free list. The *free list* is a linked list of blocks in the heap that are not currently being used, but can be used later. Each time you allocate a block, it is taken from the free list, and each time you deallocate a block, it is returned to the free list. When you want to return the block to the heap to be recycled, the pointer must be sent in as a parameter to the function that deallocates the memory. A variable with a special pointer type is normally used to hold the pointer value. Each dynamically allocated block of memory is accessed using a pointer variable which contains a pointer value that was returned by the allocation function.

For example, in C, the allocator function is called `malloc`, and the deallocator function is called `free`. In C++, the allocator is implemented as the `new` operator,

[1] Some references use the name memory pool instead of heap.

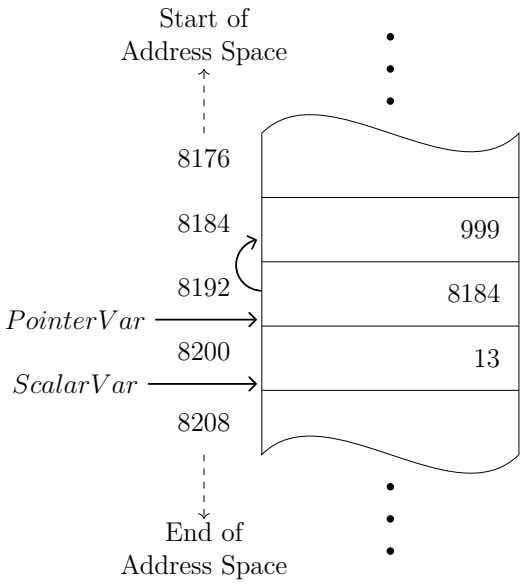


Figure 18.2: Pointer variables contain addresses that can be changed. Scalar variables are aliases for constant addresses.

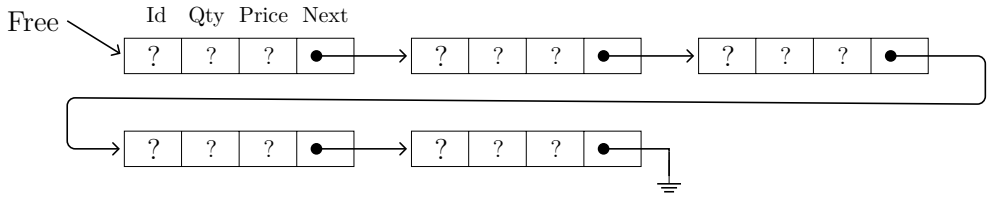


Figure 18.3: Free List

and the deallocator is implemented as the `delete` operator. It is important to remember to deallocate any block you are not using so that the heap will not run out of memory for future allocation requests. One common error that can occur when using dynamic memory is a case where memory was allocated, but not deallocated, even though it is no longer used. This error is known as a *memory leak*.

Some computer languages like Java and Go do not have a deallocate function or method. Instead, they use a technique called garbage collection, where the runtime system will keep track of the memory blocks allocated and automatically deallocate them when the blocks are no longer used. This is less efficient than using a deallocator function or method, but it makes memory leak errors impossible. Another benefit of garbage collection is that programmers do not have to think about how to ensure memory is recycled, especially when handling error cases.

In computer languages which are similar except for whether or not they use garbage collection, it is **much** easier to write and debug programs in languages with garbage

collectors. However, when using garbage collection, the generated executables are not as efficient as correctly coded programs that use explicit deallocation. Whether or not that efficiency versus ease of programming trade-off is acceptable depends on many factors. This is the reason why languages like C and C++ that use explicit memory deallocation are still used for some applications where performance of the executable is critical, even though using explicit memory deallocation requires more skill, effort, and time from programmers when writing those programs.

18.2 Storing a linked list in a matrix

To store the nodes for the linked list, a matrix (a two-dimensional array) will be used where each row is a node, and each column is a field of the record stored in the node. That matrix will be the heap that will be used for the program. Since Minimal BASIC does not have a pointer type, the normal numeric type scalar variables will be used for the pointer variables. The address they store will not be the usual memory addresses described earlier, but instead will be array index values. In this program's heap, all blocks are the same size, and each block is accessed by its index, which serves as the pointer value. This system is simple, and when a block is allocated, it is not necessary to specify the size because all blocks are the same size. Each row of the array will hold one node. Those nodes hold one record together with a next pointer. The fields of the records and the next pointer are stored in column elements in the row. The values will be accessed as a record with the fields stored in the columns. Nodes will be allocated and deallocated from a heap conceptually, but in practice rows will actually be allocated and deallocated from the array. Pointers will be used conceptually, but in practice array indices will actually be used.

Index	Identifier	Quantity	Price	Next
	0	1	2	3
0	1	2	99	2
1	4	1	100	3
2	7	8	30	1
3	0	9	9	-1
4	?	?	?	5
5	?	?	?	6
6	?	?	?	7
7	?	?	?	8
8	?	?	?	-1

Each row contains one node.

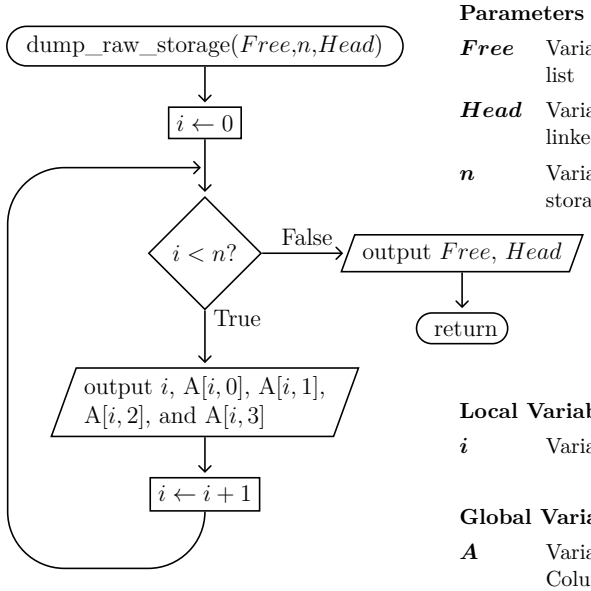
Head [0] Free [4]

Figure 18.4: Singly linked list in a matrix

For the linked list of nodes shown in figure 18.1 on page 149, one column will be needed for the Identifier, one column for the Quantity, one column for the unit Price, and finally one column will be needed for the Next pointer. All of the fields are numeric, so a matrix of numbers will work well. Remember, while the *Next* pointer is logically holding a pointer, it is really holding an array index value for the row of the array that is holding the next node in the linked list.

The linked list in figure 18.1 on page 149 has been stored in the 4×9 matrix shown in figure 18.4 on page 153. Column 0 holds the unique identifier (the key), column 1 holds the quantity, column 2 holds the unit price, and column 3 holds the link to the next node in the list. You will notice that the next value for rows 3 and 8 is set to the value -1. Since array indices cannot be negative, the value -1 is used to indicate a null pointer. A null pointer is a special pointer value that points to nowhere. It is used when you have a pointer variable but you do not want it to be pointing to any block of memory. For a next pointer of a linked list node, this value means that there is no next node. This pointer to nowhere has many names depending on what computer language you are using. In C it is called `NULL`, in Pascal it is called `nil`, in C++ it is called `nullptr`, and in Java it is called `null`. In ECMA-55 Minimal BASIC there are no pointers, but instead array indices are being used. In the last node of any singly linked list, the next pointer must contain the null pointer value to indicate that there are no more nodes in the list.

Once you have your data in the matrix, you may want to dump the matrix to verify that the contents of the matrix are correct. You can make a small procedure to do that, and then call the procedure after any changes to the linked list for debugging purposes. A flowchart for a procedure to simply dump the contents of the matrix in row order is shown in figure 18.5, and some ECMA-55 Minimal BASIC code to implement `dump_raw_storage` is shown in figure 18.23 on page 175, lines 1030 through 1120.



Parameters

Free	Variable holding index of first node in the free list
Head	Variable holding index of first node in the linked list
n	Variable holding number of elements in storage array

Local Variables

i	Variable holding loop index

Global Variables

A	Variable holding $n \times 4$ matrix of node data Column 0 is *id*, Column 1 is *qty*, Column 2 is *price*, and Column 4 is *next*

Figure 18.5: `dump_raw_storage` procedure

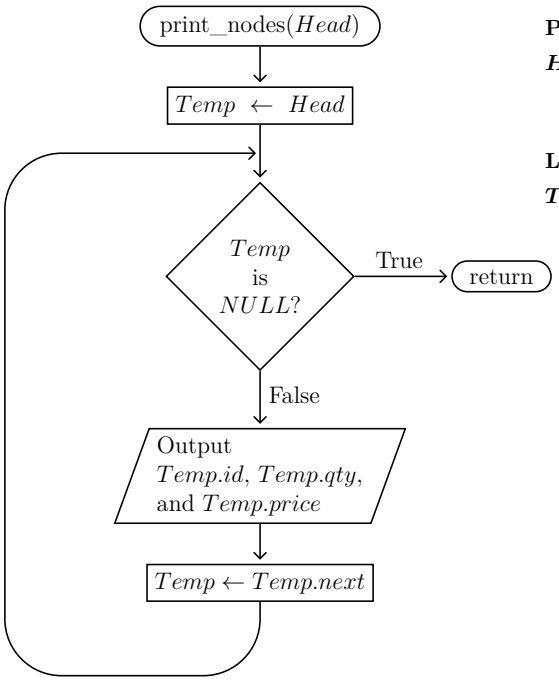
18.3 Traversing the linked list

Dumping the data with the `dump_raw_storage` function shows exactly how it is stored in the matrix, but it will not show the linked list nodes in logical order (and skip the unused nodes on the free list). To show the logical linked list in order, a different method is needed. An appropriate flowchart is shown in figure 18.6 on page 156. The ECMA-55 Minimal BASIC code to implement `print_nodes` is shown in figure 18.24 on page 176, lines 1630 through 1720.

The algorithm begins by creating a temporary pointer Temp and assigning to it the value stored in the Head pointer. Next the algorithm checks to see if Temp is NULL.[2] If Temp is NULL, there are no more nodes in the list and the algorithm ends. If Temp is not NULL, then the data in the record to which Temp points is output, then Temp is assigned the value of the next field in the record, which points to the next node in the linked list. The algorithm then loops back to the NULL test. This loop will keep processing the nodes in the linked list in the proper order until a Temp becomes NULL, and then exit. If the Head was NULL (because the linked list has no nodes), this algorithm still works. This technique of processing one node at a time and following the pointers is called traversing the linked list.[3]

[2]NULL is the null pointer.

[3]In some references, linked lists are called chains, and in those cases traversal is called following the chain.



Parameter

Head Variable holding pointer to first node in the linked list

Local Variables

Temp Variable holding pointer to current node

Figure 18.6: `print_nodes` procedure

18.4 Initializing the heap

But how can the data be loaded into the matrix in the first place? The solution to this depends on how the program should work. If you want to start with an empty list you would initialize the matrix so that all nodes are on the free list. A flowchart to do that is shown in figure 18.7, and the ECMA-55 Minimal BASIC code for `initialize_storage` is shown in figure 18.24 on page 176, lines 2090 through 2210.

On the other hand, if you want to preload some data, a different solution is needed. A flowchart to preload the data which has been stored somewhere is given in figure 18.8 on page 158. While ECMA-55 Minimal BASIC cannot access files, it does allow storing information in **`DATA`** statements and fetching those values with **`READ`** statements. ECMA-55 Minimal BASIC code to implement `load_storage` is shown in figure 18.9 on page 158.

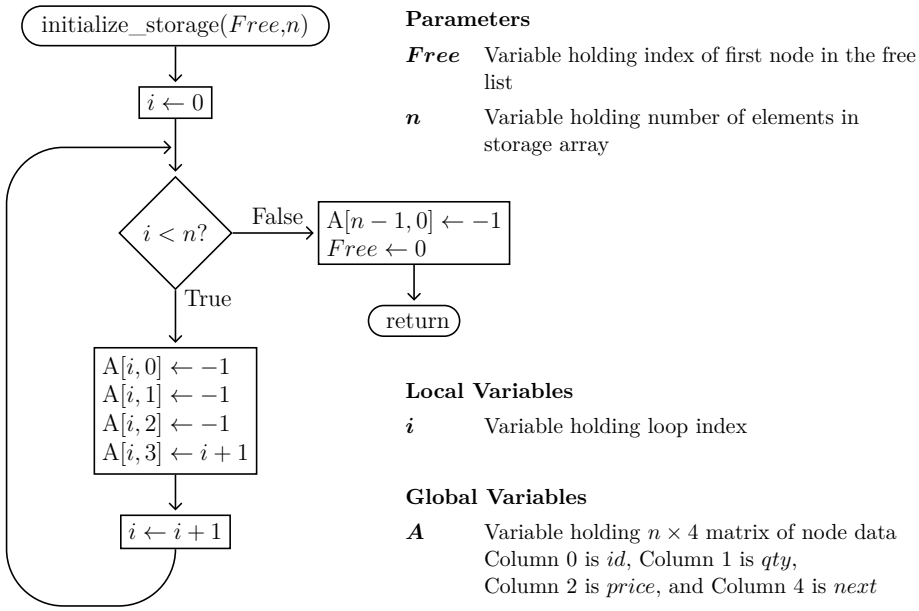
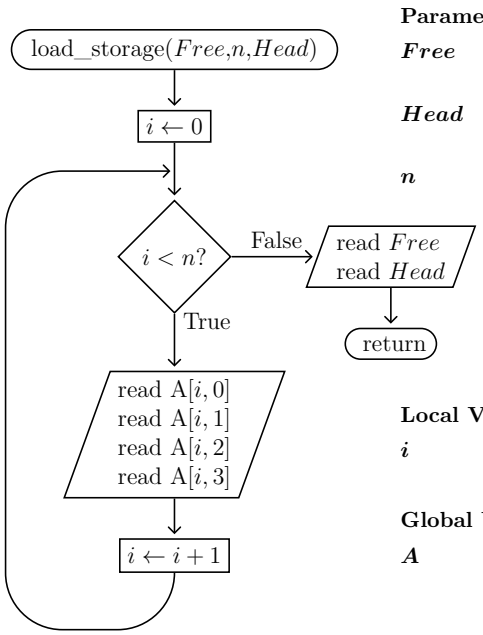


Figure 18.7: `initialize_storage` function



Parameters

Free	Variable holding index of first node in the free list
Head	Variable holding index of first node in the linked list
n	Variable holding number of elements in storage array

Local Variables

i	Variable holding loop index

Global Variables

A	Variable holding $n \times 4$ matrix of node data Column 0 is *id*, Column 1 is *qty*, Column 2 is *price*, and Column 4 is *next*

Figure 18.8: `load_storage` function

```
1000 REM LOAD_STORAGE
1010 REM N IS THE NUMBER OF ELEMENTS IN ARRAY A WHICH IS THE HEAP
1020 FOR I=0 TO N-1
1030 READ A(I,0),A(I,1),A(I,2),A(I,3)
1040 NEXT I
1050 READ H,F
1060 RETURN
```

Figure 18.9: BASIC code for load_storage

18.5 Accessing an individual node

A common task for any linked list is to print the data in a node if you know the primary key. A primary key is a unique value and it is an error for the same key to occur more than once in the list. In the example used in this chapter, the primary key is the identifier field. To print the data in a node for a specified identifier, the index of the row in the matrix where the node with that identifier is stored must be found by traversing the linked list. The algorithm must follow the links in the linked list starting with the head until the node with the correct identifier is found. This is quite similar to `print_nodes`, but only the data for the node matching the identifier is shown, and once that is output, there is no need to continue following the links in the linked list.

A function to use an identifier value to find the node is give in figure 18.10. The `find_node` function does not actually display any data, but instead just returns the pointer to the node, or NULL if there is no node with that identifier value in the linked list. For a program in ECMA-55 Minimal BASIC, that pointer is the index where the record containing that node is stored in our matrix. Once you have a pointer to the node, printing the fields in the node is straight-forward. The ECMA-55 Minimal BASIC code for `find_node` is shown in figure 18.23 on page 175, lines 1130 through 1240.

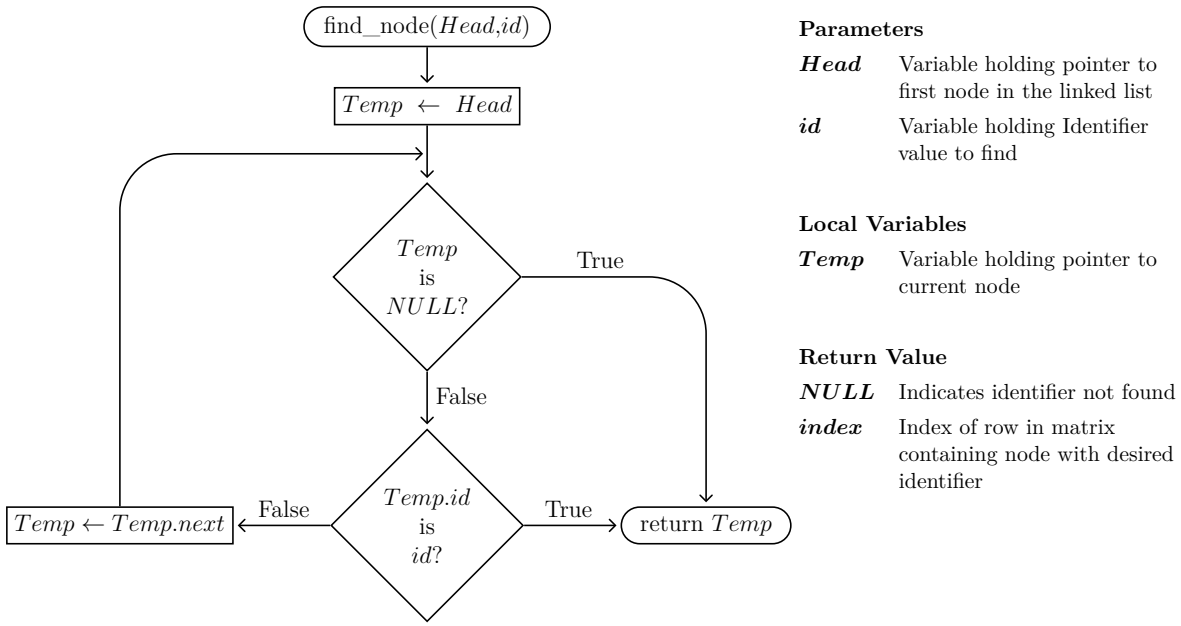
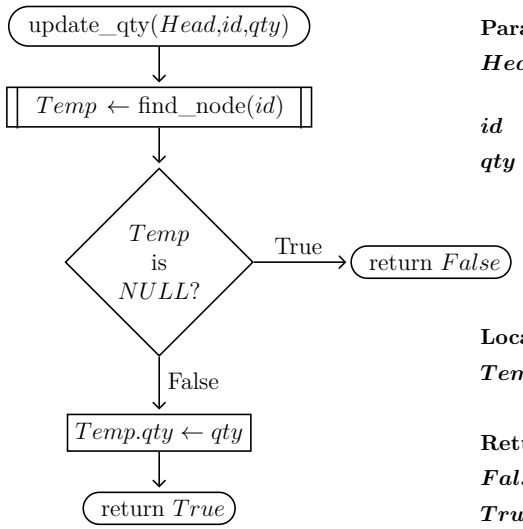


Figure 18.10: `find_node` function

18.6 Updating field values of a node

A pointer to a node allows updating the values in data fields of that node. The primary key value must not be updated, so in our example the identifier field must not be updated. Obviously the pointer to the next node must not be updated either because that would break the linked list. However, any other fields can be updated. In the example in this chapter, this means that the quantity and unit price fields can be updated. A flowchart to update the quantity is shown in figure 18.11, and the ECMA-55 Minimal BASIC code to implement `update_qty` is shown in figure 18.23 on page 175, lines 1350 through 1480. A very similar flowchart to update the unit price is shown in figure 18.12 on page 161, and the ECMA-55 Minimal BASIC code to implement `update_price` is shown in figure 18.23 on page 175, lines 1490 through 1620. Both of these call the `find_node` function to get a pointer to the node that will be updated. Since `find_node` might not succeed, it follows that `update_qty` and `update_price` might not succeed either. The program must be careful to check for these errors and report them back to the user if they occur.



Parameters

Head	Variable holding pointer to first node in the linked list
id	Variable holding identifier value to update
qty	Variable holding quantity value to update

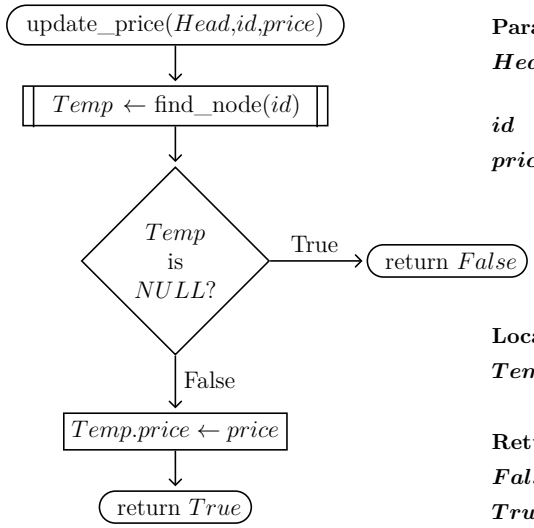
Local Variables

Temp	Variable holding pointer to current node

Return Values

False	Update failed, identifier not found
True	Update succeeded

Figure 18.11: `update_qty` function



Parameters

Head	Variable holding pointer to first node in the linked list
id	Variable holding identifier value to update
price	Variable holding price value to update

Local Variables

Temp	Variable holding pointer to current node

Return Values

False	Update failed, identifier not found
True	Update succeeded

Figure 18.12: `update_price` function

18.7 Appending a node to a linked list

Another common operation on linked lists is adding nodes to the end of the list. Achieving this goal can be tricky because there are several special cases to consider:

1. The linked list is empty.
2. The free list is empty.
3. The identifier specified is already used for a node in the list.

The `append_node` function will take a node and remove it from the free list and then add it to the end of the linked list pointed to by the head pointer. Finally it will fill in the identifier, quantity, and unit price values. Appending can fail if the free list has no nodes. Appending also must fail if a node with the same primary key is already in the linked list. This error condition is called a *duplicate key*. Finally, if the linked list is empty, the head pointer must be updated to point to the node that was removed from the free list to create a new linked list.

A series of diagrams showing logically the operation of appending a linked node to a linked list in the common case where the linked list and the free list are not empty should help explain the concept. The example begins with the linked list shown in figure 18.1 on page 149 and an new node with an identifier of 9, a quantity of 2, and a unit price of 50 must be appended to the linked list. The first step would be to find a pointer to the last node on the linked list. The second step would be to allocate a node from the free list. The final step is to add that allocated node after to the last node on the linked list and fill in that newly allocated node's data fields.

The linked list pointed to by *Head* must be traversed in order to find the last node. This is done by using two pointers, the *Current* pointer to the current node, and the *Previous* pointer which points to the previous node. When the *Current* pointer becomes NULL after traversing the last node, the *Previous* pointer will be left pointing to the last node in the linked list. These steps are shown in figure 18.13 on page 163.

The free list is always accessed using the *Free* pointer, and to get a new node the first node in the free list is taken and is used as the new node to add to the end of the linked list. This is done by using a temporary pointer for the newly allocated node called *Current*, and assigning to it the value of *Free*. Then the next step is to remove that node from the free list by first updating the *Free* pointer to point to the node after *Current*, and then setting the *Current* node's next pointer to NULL. These steps are shown in figure 18.14 on page 164.

Now that *Current* points to an allocated node removed from the free list, and we have a *Previous* pointer that points to the last node in the linked list, the *Current* node can finally be appended to the linked list. The next field of the node pointed

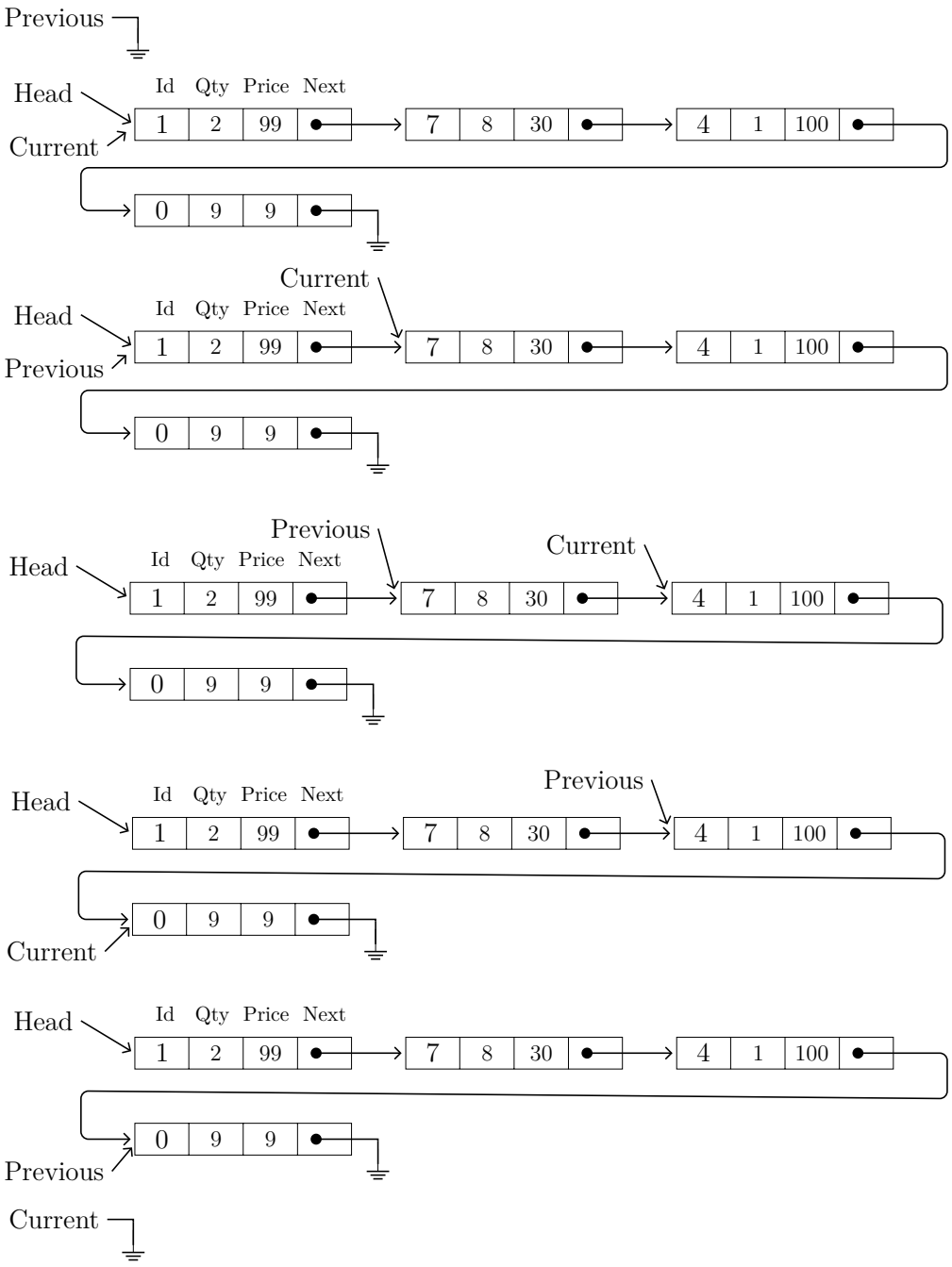


Figure 18.13: Finding the last node in the linked list

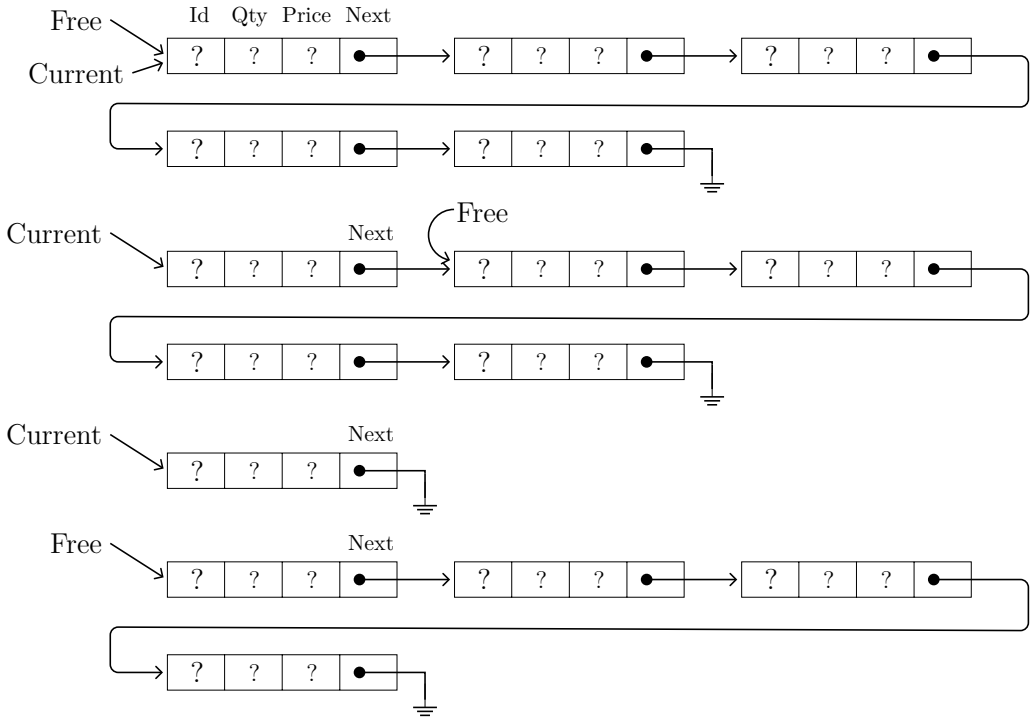


Figure 18.14: Allocating a node from the free list

to by *Previous* is assigned the value of the *Current* pointer, adding the node just taken from the free list to the end of the linked list pointed to by *Head*. This is shown in figure 18.15.

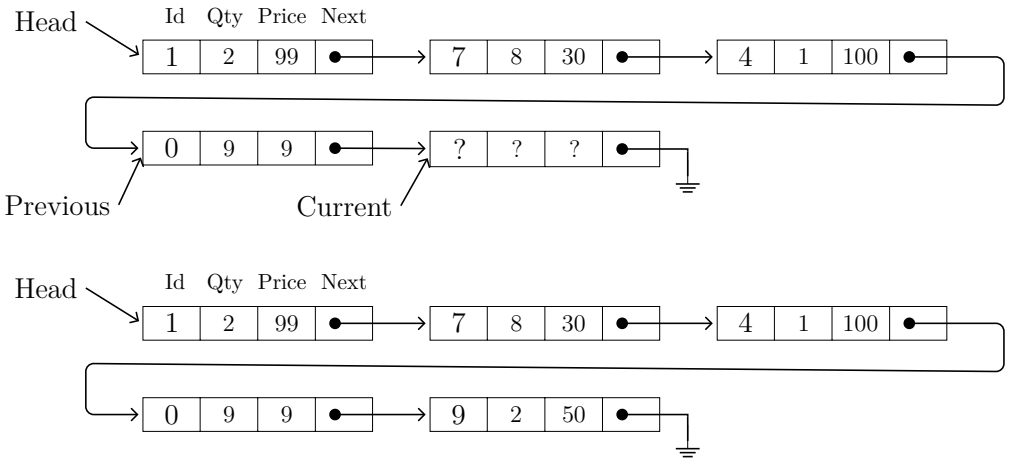


Figure 18.15: Appending the node with identifier 9 to the linked list

The last step to do to complete the append operation is to fill in the identifier, quantity, and unit price fields of the *Current* record with the values 9, 2, and 50

respectively. Now the *Current* and *Previous* pointers are not needed, and the node has been moved from the free list to the end of the linked list pointed to by *Head* and the data fields have been updated, so the append is complete and the result is shown in figure 18.15 on page 164.

While the diagrams make the append concept reasonably clear, a detailed algorithm is needed to ensure everything is done step-by-step and that all of the special cases are handled correctly. The flowchart in figure 18.16 on page 166 shows an appropriate algorithm called `append_node`, and the ECMA-55 Minimal BASIC code for `append_node` is shown in figure 18.24 on page 176, lines 1730 through 2080.

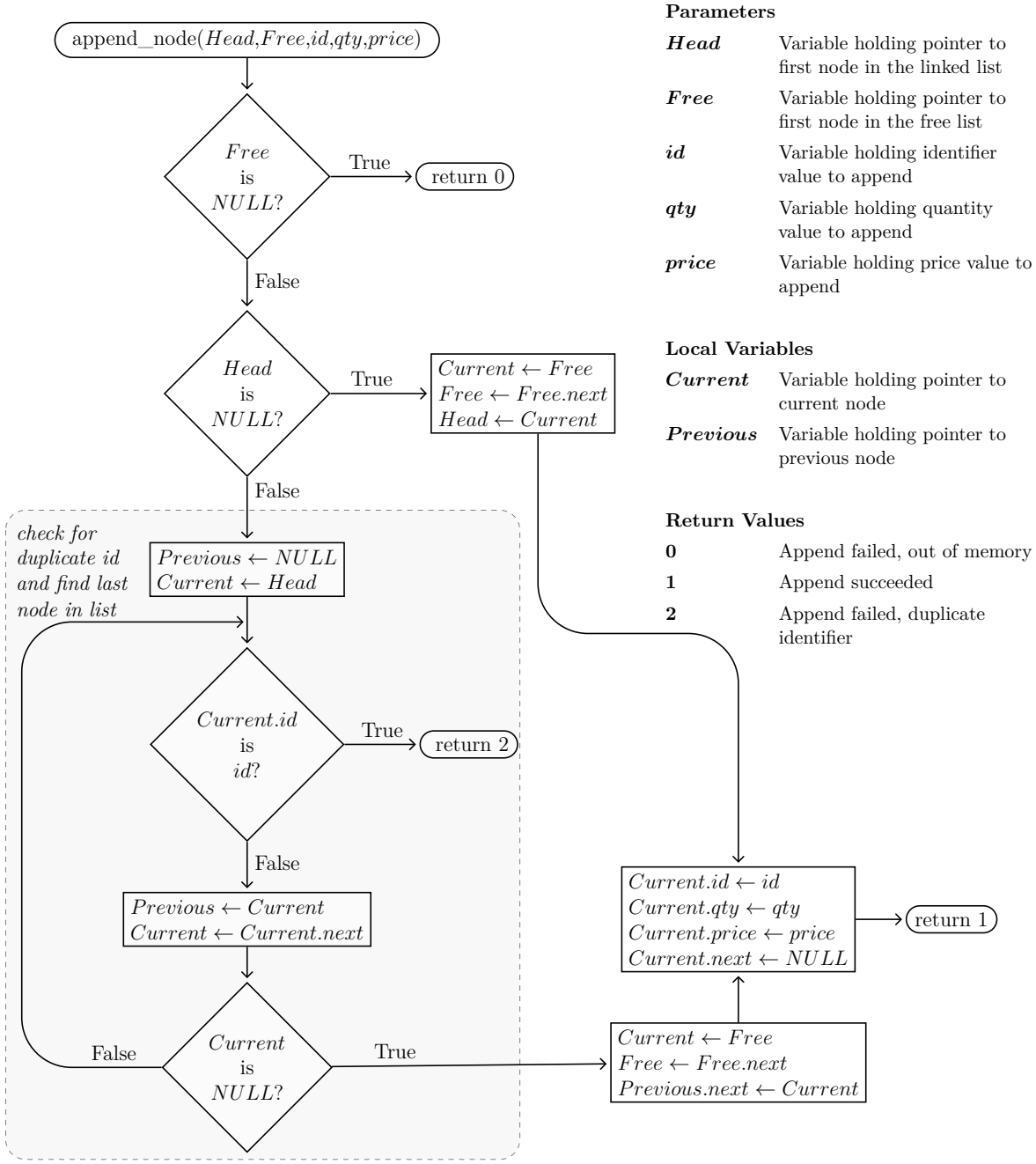


Figure 18.16: `append_node` function

18.8 Removing a node from a linked list

Another common operation on linked lists is deleting nodes from the list. Achieving this goal can be tricky because there are several special cases to consider:

1. The linked list is empty.
2. The identifier specified is not used by any node in the linked list.
3. The node to remove is the first node in the linked list.
4. The node to remove is the only node in the linked list.

A series of diagrams which show an example of deleting a node from a linked list should make the algorithm easier to understand. The first step is to search the linked list to find the node to remove, and its predecessor. A pointer to its predecessor node is needed so that the program can update that predecessor node's next pointer to point to the node that comes after the node being deleted. The node pointer of the node after the node being deleted is in the node being deleted's next field. The way to get pointers to the node which is to be deleted and its predecessor is to use two pointers. One of those pointers points to the current node being visited, and the other one points the last node that was visited. The last node visited is the predecessor node of the current node. Each time the next node is visited, both pointers are updated. Once the node whose identifier matches the identifier to be removed is found, both pointers are set to the correct nodes. This step-by-step process is shown in figure 18.17 on page 169, where the identifier being sought is zero. Once it is found, that node is removed from the linked list by updating the next pointer of the predecessor node to have the same value as the next pointer of the node with identifier zero. The node with identifier zero then has its next pointer set to NULL. At that point the linked list is updated, but the node with identifier zero is not on the free list yet.

When a node is removed, its storage should be recycled so it can be used later when a node is appended to the linked list. The way this recycling occurs is that deleted nodes are prepend to the front of the free list. Prepending nodes is easier than adding them to the end of the list. That is because it is not necessary to scan the free list to find the last node. The deleted node is added to the front of the free list instead of adding it to the end of the free list. The node with identifier zero will have its next field updated to have the value of the free list pointer, which points to the front of the free list. This creates a link from the end of the deleted node to the head of the free list. The last step is to update the head of the free list to point to the newly deleted node with identifier zero. At that point, the deleted node has been prepended to the free list. This technique is shown in figure 18.18 on page 170.

Describing all of these steps in words is rather long, but the technique itself is not too difficult. As shown in figure 18.19 on page 170, the desired result has been achieved.

While the diagrams make the delete concept reasonably clear, a detailed algorithm is needed to ensure everything is done step-by-step and that all of the special cases are handled correctly. The flowchart in figure 18.20 on page 171 shows an appropriate algorithm called `delete_node`. The ECMA-55 Minimal BASIC code for `delete_node` is shown in figure 18.25 on page 177, lines 2220 through 2470.

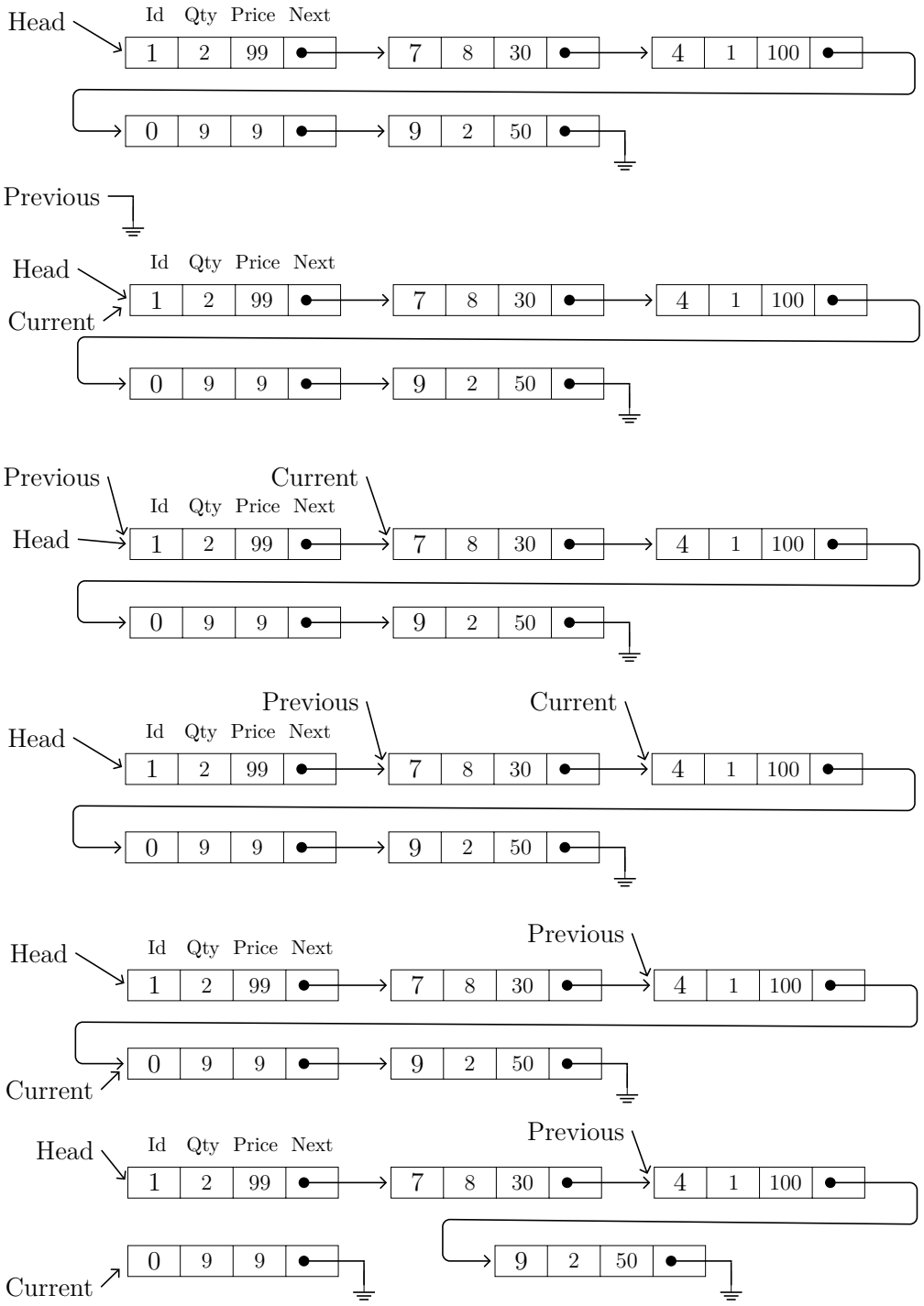


Figure 18.17: Finding and removing node with identifier 0 from linked list

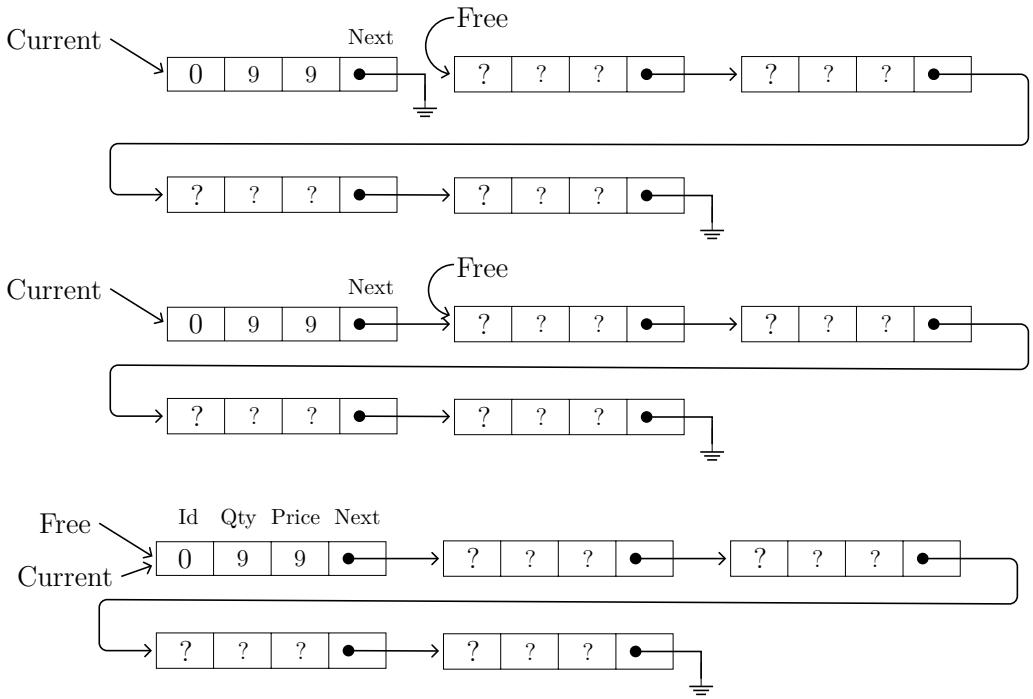


Figure 18.18: Prepend deleted node to free list

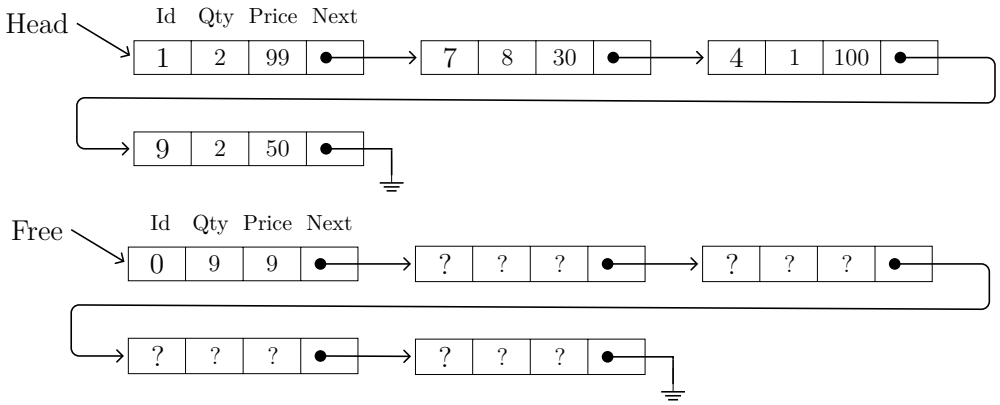


Figure 18.19: Linked list and free list after delete

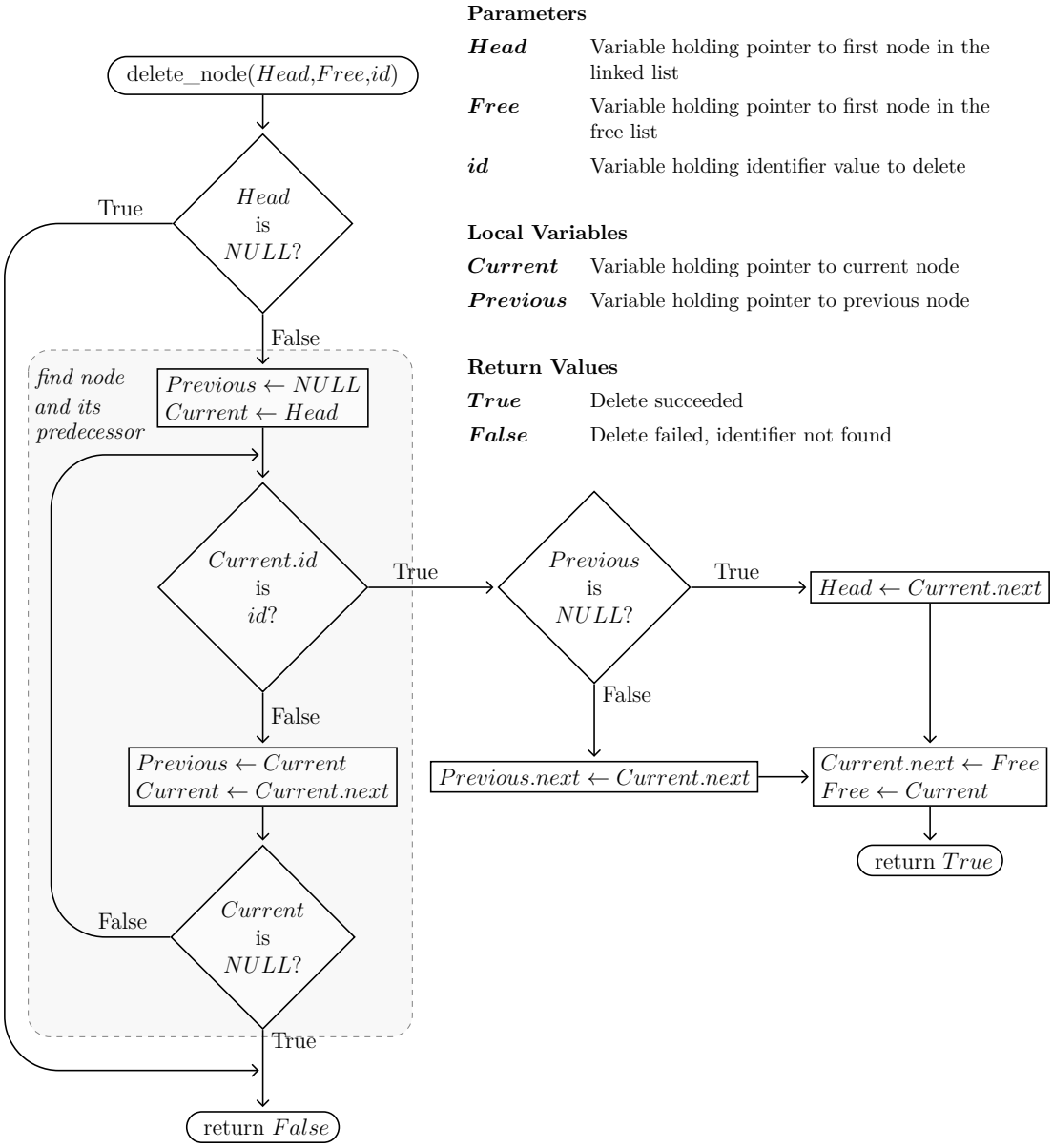


Figure 18.20: `delete_node` function

18.9 Implementation in ECMA-55 Minimal BASIC

The theory behind dynamic memory and singly linked lists has been presented in this chapter, together with the necessary details to implement a simple dynamic memory manager using a matrix. The algorithms for common operations on linked lists have been presented with detailed flowcharts as well. Once you put all of that knowledge to work, you will be able to write programs which use singly linked lists. A complete, working example program that demonstrates everything is shown in figures 18.21 through 18.25 on pages 173-177. The example program has a menu-driven interface, and implements every algorithm except `load_storage`. The sample program uses separate subroutines for each of the seven different operations, as well as one for `initialize_storage`. An index of subroutine locations is provided in table 18.1.

Operation	Lines	BASIC Page	Flowchart Page
`dump_raw_storage`	1030-1120	175	154
`find_node`	1130-1240	175	159
`update_qty`	1350-1480	175	160
`update_price`	1490-1620	175	161
`print_nodes`	1630-1720	176	156
`append_node`	1730-2080	176	166
`initialize_storage`	2090-2210	176	157
`delete_node`	2220-2470	177	171

Table 18.1: Singly linked list demo subroutine locations

```
10 REM LINKED LIST DEMO
20 REM ARRAY INDICES ARE ZERO-BASED
30 OPTION BASE 0
40 REM ARRAY A HAS 9 ROWS AND 4 COLUMNS AND IS THE HEAP
50 REM COLUMN 0 IS IDENTIFIER
60 REM COLUMN 1 IS QUANTITY
70 REM COLUMN 2 IS UNIT PRICE
80 REM COLUMN 3 IS NEXT
90 REM F IS HEAD OF FREE LIST IN HEAP
100 REM H IS HEAD OF LINKED LIST OF NODES
110 REM N IS NUMBER OF ROWS IN ARRAY A (NUMBER OF BLOCKS IN HEAP)
120 LET N=9
130 DIM A(8,3)
140 REM START OF MAIN
150 REM ******* INITIALIZE HEAP *******
160 GOSUB 2090
170 REM ******* SET HEAD TO NULL *******
180 LET H=-1
190 REM ******* SHOW MENU *********
200 GOSUB 2480
210 ON C GOTO 230,270,370,840,540,680,930,980
220 REM
230 REM ******* SHOW LIST *********
240 REM
250 GOSUB 1630
260 GOTO 190
270 REM ******* LOOKUP A NODE IN LIST *********
280 PRINT "WHAT IS THE NODE ID";
290 INPUT W
300 GOSUB 1130
310 IF T=-1 THEN 340
320 GOSUB 1250
330 GOTO 190
340 PRINT "NODE ";W;" NOT FOUND"
350 GOTO 190
360 REM
370 REM ******* APPEND A NODE TO LIST *********
380 REM
390 PRINT "WHAT IS THE NODE ID";
400 INPUT I9
410 PRINT "WHAT IS THE QUANTITY";
420 INPUT Q9
430 PRINT "WHAT IS THE PRICE";
440 INPUT P9
450 GOSUB 1730
460 ON R+1 GOTO 490,510,470
470 PRINT "ID ";I9;" IS ALREADY IN THE LIST"
480 GOTO 820
490 PRINT "OUT OF MEMORY"
500 GOTO 820
510 PRINT "APPEND OF ID=";I9;", QUANTITY=";Q9;", PRICE=";P9;" SUCCESSFUL"
520 GOTO 190
530 REM
```

Figure 18.21: BASIC code for singly linked lists (1 of 5)

```
540 REM ******* UPDATE QUANTITY OF NODE IN LIST *********
550 REM
560 PRINT "WHAT IS THE NODE ID";
570 INPUT I0
580 PRINT "WHAT IS THE NEW QUANTITY";
590 INPUT Q0
600 PRINT "UPDATE ID=";I0;" QUANTITY=";Q0;
610 GOSUB 1350
620 IF R=1 THEN 650
630 PRINT " FAILED"
640 GOTO 660
650 PRINT "SUCCEEDED"
660 GOTO 190
670 REM
680 REM ******* UPDATE PRICE OF NODE IN LIST *********
690 REM
700 PRINT "WHAT IS THE NODE ID";
710 INPUT I0
720 PRINT "WHAT IS THE NEW PRICE";
730 INPUT P0
740 GOSUB 1490
750 PRINT "UPDATE ID=";I0;" PRICE=";P0;
760 IF R=1 THEN 790
770 PRINT " FAILED"
780 GOTO 800
790 PRINT " SUCCEEDED"
800 GOTO 190
810 REM
820 GOTO 190
830 REM
840 REM ******* DELETE A NODE FROM LIST *********
850 REM
860 PRINT "WHAT IS THE NODE ID";
870 INPUT I9
880 GOSUB 2220
890 IF R=1 THEN 900
894 PRINT "NODE ";I9;" NOT FOUND"
898 GOTO 190
900 PRINT "DELETE OF ID=";I9;" SUCCESSFUL"
910 GOTO 190
920 REM
930 REM ******* RAW DUMP OF HEAP INFORMATION (DEBUG) ********
940 REM
950 GOSUB 1030
960 GOTO 190
970 REM
980 REM ******* QUIT ********
990 REM
1000 STOP
1010 REM END MAIN
```

Figure 18.22: BASIC code for singly linked lists (2 of 5)

```
1020 REM
1030 REM ****************************
1040 REM * DUMP_RAW_STORAGE PROCEDURE *
1050 REM ****************************
1060 FOR I=0 TO N-1
1070 PRINT I;":";A(I,0);",";A(I,1);
1080 PRINT ",";A(I,2);",";A(I,3)
1090 NEXT I
1100 PRINT "HEAD IS";H
1110 PRINT "FREE IS";F
1120 RETURN
1130 REM **********************
1140 REM * FIND_NODE FUNCTION *
1150 REM **********************
1160 REM W IS THE IDENTIFIER WE WANT
1170 REM RETURNS T WHICH IS ARRAY INDEX OF NODE
1180 REM WITH ID W, OR -1 IF NODE NOT FOUND
1190 LET T=H
1200 IF T=-1 THEN 1240
1210 IF A(T,0)=W THEN 1240
1220 LET T=A(T,3)
1230 GOTO 1200
1240 RETURN
1250 REM *****************
1260 REM * SHOW ONE NODE *
1270 REM *****************
1280 REM T IS THE POINTER TO THE NODE WHOSE DATA WILL BE DISPLAYED
1290 PRINT "ITEM FOUND IN SLOT";T
1300 PRINT "IDENTIFIER:";A(T,0)
1310 PRINT "QUANTITY  :";A(T,1)
1320 PRINT "UNIT PRICE:";A(T,2)
1330 PRINT "NEXT      :";A(T,3)
1340 RETURN
1350 REM **********************
1360 REM * UPDATE_QTY FUNCTION *
1370 REM **********************
1380 REM I0 IS ID
1390 REM Q0 IS QUANTITY
1400 REM R IS RETURN (0 IS FALSE, 1 IS TRUE)
1410 LET W=I0
1420 GOSUB 1130
1430 IF T=-1 THEN 1470
1440 LET A(T,1)=Q0
1450 LET R=1
1460 GOTO 1480
1470 LET R=0
1480 RETURN
1490 REM ************************
1500 REM * UPDATE_PRICE FUNCTION *
1510 REM ************************
1520 REM I0 IS ID
1530 REM P0 IS PRICE
1540 REM R IS RETURN (0 IS FALSE, 1 IS TRUE)
1550 LET W=I0
1560 GOSUB 1130
1570 IF T=-1 THEN 1610
1580 LET A(T,2)=P0
1590 LET R=1
1600 GOTO 1620
1610 LET R=0
1620 RETURN
```

Figure 18.23: BASIC code for singly linked lists (3 of 5)

```
1630 REM ************************
1640 REM * PRINT_NODES PROCEDURE *
1650 REM ************************
1660 REM T0 IS TEMP
1670 LET T0=H
1680 IF T0=-1 THEN 1720
1690 PRINT A(T0,0),A(T0,1),A(T0,2)
1700 LET T0=A(T0,3)
1710 GOTO 1680
1720 RETURN
1730 REM ***************
1740 REM * APPEND_NODE *
1750 REM ***************
1760 REM I9 IS NEW IDENTIFIER
1770 REM Q9 IS NEW QUANTITY
1780 REM P9 IS NEW UNIT PRICE
1790 REM R IS RETURN (1 IS SUCCESS, 0 IS OOM, 2 IS DUPLICATE KEY)
1800 REM T1 IS POINTER TO CURRENT NODE
1810 REM T2 IS POINTER TO PREVIOUS NODE
1820 IF F<>-1 THEN 1850
1830 LET R=0
1840 GOTO 2080
1850 IF H=-1 THEN 2040
1860 LET T2=-1
1870 LET T1=H
1880 IF A(T1,0)<>I9 THEN 1910
1890 LET R=2
1900 GOTO 2080
1910 LET T2=T1
1920 LET T1=A(T1,3)
1930 IF T1=-1 THEN 1950
1940 GOTO 1880
1950 LET T1=F
1960 LET F=A(F,3)
1970 LET A(T2,3)=T1
1980 LET A(T1,0)=I9
1990 LET A(T1,1)=Q9
2000 LET A(T1,2)=P9
2010 LET A(T1,3)=-1
2020 LET R=1
2030 GOTO 2080
2040 LET T1=F
2050 LET F=A(F,3)
2060 LET H=T1
2070 GOTO 1980
2080 RETURN
2090 REM **********************
2100 REM * INITIALIZE_STORAGE *
2110 REM **********************
2120 REM N IS THE NUMBER OF ELEMENTS IN ARRAY A WHICH IS THE HEAP
2130 FOR I=0 TO N-1
2140 LET A(I,0)=-1
2150 LET A(I,1)=-1
2160 LET A(I,2)=-1
2170 LET A(I,3)=I+1
2180 NEXT I
2190 LET A(N-1,3)=-1
2200 LET F=0
2210 RETURN
```

Figure 18.24: BASIC code for singly linked lists (4 of 5)

```
2220 REM ***************
2230 REM * DELETE_NODE *
2240 REM ***************
2250 REM I9 IS IDENTIFIER OF NODE TO DELETE
2260 REM R IS RETURN (1 IS SUCCESS, 0 IS FAILURE)
2270 REM T1 IS POINTER TO CURRENT NODE
2280 REM T2 IS POINTER TO PREVIOUS NODE
2290 REM EACH ROW IS DATA FOR ONE NODE
2300 IF H=-1 THEN 2460
2310 LET T2=-1
2320 LET T1=H
2330 IF A(T1,0)=I9 THEN 2380
2340 LET T2=T1
2350 LET T1=A(T1,3)
2360 IF T1<>-1 THEN 2330
2370 GOTO 2460
2380 IF T2=-1 THEN 2410
2390 LET A(T2,3)=A(T1,3)
2400 GOTO 2420
2410 LET H=A(T1,3)
2420 LET A(T1,3)=F
2430 LET F=T1
2440 LET R=1
2450 GOTO 2470
2460 LET R=0
2470 RETURN
2480 REM ***********************************
2490 REM * SHOW MAIN MENU AND GET A CHOICE *
2500 REM ***********************************
2510 PRINT "    LINKED LIST MANAGER"
2520 PRINT
2530 PRINT "1.  SHOW LIST"
2540 PRINT "2.  LOOKUP A NODE IN LIST"
2550 PRINT "3.  APPEND A NODE TO LIST"
2560 PRINT "4.  DELETE A NODE FROM LIST"
2570 PRINT "5.  UPDATE QUANTITY OF NODE IN LIST"
2580 PRINT "6.  UPDATE PRICE OF NODE IN LIST"
2590 PRINT "7.  RAW DUMP OF HEAP INFORMATION (DEBUG)"
2600 PRINT "8.  QUIT"
2610 PRINT
2620 PRINT "WHAT DO YOU WANT TO DO";
2630 INPUT C
2640 IF INT(C)=C THEN 2680
2650 PRINT "INVALID INPUT!"
2660 PRINT
2670 GOTO 2510
2680 IF C<1 THEN 2650
2690 IF C>8 THEN 2650
2700 RETURN
2710 END
```

Figure 18.25: BASIC code for singly linked lists (5 of 5)

Exercises

1. In this chapter, only one linked list pointed to by a special Head pointer has been used. Actually, multiple linked lists can exist, all using one array for storage and sharing one common free list. The update and delete flowcharts do not need to be updated for this. Why will they still work?

2. If the linked list needed to be sorted, how could it be done? Draw a flowchart for an appropriate algorithm, and then write a ECMA-55 Minimal BASIC program and test it.

Chapter 19

Summary

This document has introduced you to all of the statements used in the ECMA-55 Minimal BASIC dialect of the BASIC language. Solutions for many mathematical problems can be written using this dialect, and this dialect is for the most part a subset of the popular classic microcomputer BASIC languages used such as BASIC-80, BASICA, GWBASIC, etc. With the strong foundation you now have after learning everything in this document, learning any of those other dialects will not be difficult. Those dialects add many exciting features such as comprehensive support for strings and string arrays, access to files on the disk, and more. Some of them even give you the ability to write programs using graphics and sound. But please don't think you are limited to just the BASIC computer language. Flowcharts are language-agnostic, and many languages available today are easy to learn now that you understand the BASICs of programming.

Appendix A

The bas55 Interpreter

This appendix describes how to use the **bas55** ECMA-55 Minimal BASIC interpreter program from Jorge Giner Cordero. The interpreter is available at his web site **https://jorgicor.niobe.org/bas55/** as a pre-compiled version for **Microsoft® Windows®** or as a version for **POSIX®** systems like **Linux®**, **FreeBSD®**, etc. that you can build from source. Alternatively, the source code can be found on GitHub at **https://github.com/jorgicor/bas55/releases**.

Mr. Cordero's web site has a great manual, but here is a simple sample session so you can see what using **bas55** is like, and learn the most important commands:

- **RUN** to run the current program.
- **LIST** to see the source code of the current program in proper order.
- **LOAD** to clear the current program, if any, and load the program stored in the specified file.
- **NEW** to clear the current program.
- **RENUM** to renumber the current program.
- **SAVE** to save the current program to the specified file.
- **QUIT** to exit the interpreter, losing any current program.

The *current program* is the program that the interpreter has loaded into memory. When you first start the interpreter, no program is loaded, but you can either type in a program or use the **LOAD** command to load one from a file. Any changes you make to the program, including using **RENUM**, are done to the current program. When you exit **bas55**, those changes are lost. To keep such changes permanently, you need to use the **SAVE** command to save the current program to a file. The **bas55** interpreter does warn you if you exit without saving and asks if you want to discard the current program. If you answer **y**, the current program is lost and bas55 exits. If you say **n**, then the quit is canceled and you have a chance to use

the **SAVE** command. If you already saved your file and did not make any changes after that, then quit does not ask any questions and **bas55** just exits immediately.

If you try to save to an existing file, **bas55** will tell you the file already exists and ask if you want to *overwrite* the existing file. If you answer **y**, the file's existing content is discarded and the current program is stored in the file, replacing the old program. Any old data in the file is lost. If you answer **n**, then the **SAVE** does nothing. This is a safety feature so you don't accidentally *overwrite* an important program. So you can **SAVE** on top of the old version, but you must confirm that is what you really want to do before **bas55** will let you do it.

Any line that begins with a line number is added to the program. If a line with the same number already existed, that old line is replaced by the new one. If a line number is typed without anything after it, then the line with that number in the program is removed. The `Ready.` message tells you that the interpreter is now ready and waiting for you to type something.

```
bas55 1.19

This is free software: you are free to change and redistribute it,
but there is NO WARRANTY. Type LICENSE to show the details.

Type HELP for a list of allowed commands.
Ready.
10 PRINT "BASIC IS EASY!"
20 END
RUN
BASIC IS EASY!
Ready.
LIST
10 PRINT "BASIC IS EASY!"
20 END
Ready.
SAVE "BIE.BAS"
Ready.
NEW
Ready.
LIST
Ready.
LOAD "BIE.BAS"
Ready.
LIST
10 PRINT "BASIC IS EASY!"
20 END
Ready.
20 PRINT "IS IT BREAK TIME YET?"
RUN
20: error: program must have an END statement
Ready.
30 END
RUN
BASIC IS EASY!
IS IT BREAK TIME YET?
Ready.
QUIT
Discard current program? (y/n)Y
```

Figure A.1: Sample bas55 session

NOTE 1: The **bas55** interpreter initializes all numeric variables to zero and all string variables to empty strings. This means that if you read an unintialized variable, the interpreter will use a zero or empty string for numeric and string variable values respectively, and then continue running the program. As of version 1.16, **bas55** will issue a helpful warning for this issue, but the program will keep running.

Appendix B

The ecma55 Compiler

This appendix describes how to use the **ecma55** ECMA-55 Minimal BASIC compiler program written by the author of this book. The compiler is available at the web site **http://sourceforge.net/projects/buraphakit/** and is distributed as source code. It can be built on x86-64 Linux® systems, and the programs it creates run on x86-64 Linux® systems.

After you download the source code, you will need to build it. The latest version at the time this document was last updated was `MinimalBASIC-2.40.tar.xz` but you might find a newer version has been released. If so, just change the version number in these instructions and they should still work.

```
xz -dc MinimalBASIC-2.40.tar.xz | tar -x -v -f -
cd MinimalBASIC-2.40
make -fMakefile.gcc
```

To install the software so you can use it does not require root powers. Just manually copy the files to your personal binary directory.

```
mkdir -p ${HOME}/bin
cp -a ecma55 BASICC BASICCS BASICCW ${HOME}/bin/
```

If your `PATH` does not include `${HOME}/bin`, then you will need to add it in your shell's initialization file. For GNU `bash`, the `${HOME}/.profile` file works great and you can add a line like this:

```
export PATH="${PATH}:${HOME}/bin"
```

Then log out and log back in and things should work fine for you.

However, things are nicer if you can install them for all users. To install the software so all users can use it you need to have `root` user power with either the traditional

`su` command or with the `sudo` command favored by many new distributions. Get a `root bash` shell, then execute the following command:

```
make -fMakefile.gcc install
```

Using that command will put the binaries in `/usr/local/bin` and the manual pages in `/usr/local/man`.

While the defaults work well for most people, it is possible to customize the installation quite a bit. You can specify `PREFIX` to install into an alternate location like this:

```
make -fMakefile.gcc install PREFIX=/opt/ecma55
```

Using that command will put the binaries in `/opt/ecma55/bin` and the manual pages in `/opt/ecma55/man`.

Some systems want man pages in `/usr/share/man` and you can achieve that as part of a standard installation with this command:

```
make -fMakefile.gcc install MANDIR=/usr/share/man/man1 PREFIX=/usr
```

Here is a sample session. Normally the **`TEST.BAS`** program would be entered using a text editor like **`vi`**, but for this very short program the **`cat`** program is used instead. The '`$`' (dollar sign) is the system prompt from the Bourne-compatible shell, which will be GNU `bash` on most Linux® systems.

```
$cat <<EOF >TEST.BAS
10 PRINT "BASIC IS EASY!"
20 PRINT "IS IT BREAK TIME YET?"
30 END
EOF
$BASICC TEST.BAS
$./TEST
BASIC IS EASY!
IS IT BREAK TIME YET?
$
```

Appendix C

BASIC Statements

This appendix has a list of all of the statements in ECMA-55 Minimal BASIC. Each entry in the list briefly describes the syntax and semantics of the statement. All keywords and expressions must be surrounded by at least one space except at the end of the line where no trailing space is required.

DATA

The **DATA** statement is used to contain values that will be later used by the **READ** statement. The form of the **DATA** statement is:

```
DATA value1,value2,...
```

After the **DATA** keyword, a list of one or more comma-delimited values must be present. The values can be numeric literal values, double-quoted string literal values, or unquoted strings. You should not use unquoted strings in new programs. See the ECMA-55 Minimal BASIC standard for details of unquoted strings if you encounter them in existing code. At program startup, the logical read pointer points the first value in the lowest-numbered **DATA** statement. Each time a value is read with the **READ** statement, this pointer advances. It can be reset to the first value of the first[1] **DATA** statement in the program using the **RESTORE** statement. See chapter 11 for more information and example programs.

[1]The first **DATA** statement is the **DATA** statement which has the lowest integral value for a line number.

DEF

The **DEF** statement is used to define a user-defined function of one numeric variable or a pseudo-constant. The form of the **DEF** statement is:

```
DEF FNx = numeric_expression
```

or

```
DEF FNx (parameter) = numeric_expression
```

where x is one of the single letters between A and Z inclusive, and a parameter is a scalar numeric variable. The first form of the **DEF** statement is rarely used since the it can easily be replaced by a literal value, but can serve as a pseudo-constant since it cannot be changed later, unlike a normal global scalar variable.

In the second form of the **DEF** statement, the parameter is a local variable that is only visible in the following *numeric_expression*, and it is distinct from any global variable of the same name. There is no way to use the global variable with the same name as the parameter in the *numeric_expression* of a **DEF** statement, although all other global variables can be used.

Either form used to define a user-defined function must occur before any call to it occurs to the corresponding user-defined function in the program. Normally, the **DEF** statements, like **DIM** statements, occur at the beginning of the program. The **DEF** statement declares the function, but otherwise does nothing when it is executed. A user-defined function may call any built-in or user-defined function except itself. Note, however, that any referenced user-defined function must have already been defined. A user-defined function may not be re-defined. See chapter 17 for more information and example programs.

DIM

The **DIM** statement is used to specify non-default sizes of numeric arrays. The form of the **DIM** statement is:

```
DIM declaration1,declaration2,...
```

and declarations come in two forms:

```
X(maxsubscript)
X(maxsubscript1,maxsubscript2)
```

where X is one of the 26 possible single-letter array variable names. The first form is for one-dimensional arrays (vectors), and the second form is for two-dimensional arrays (matrices). After the **DIM** keyword, a list of one or

more comma-delimited array declarations is specified. Each array declaration is an array name followed by a left parenthesis, then a non-negative integer value. For a one-dimensional array, this is followed immediately by a right parenthesis. For a two-dimensional array, this is followed by a comma, another non-negative integer value, and then a right parenthesis. The highest subscript permitted for the arrays is specified in the **DIM** statement. The lowest subscript is zero unless **OPTION BASE 1** was specified earlier in the program. The maximum possible value of an array dimension is implementation-defined. See chapter 10 for more information and example programs.

END

The **END** statement is used to specify the end of the source program, and it must occur exactly once in the program on the last line of the source which must have a line number higher than all other line numbers in the source program. The general form of the **END** statement is:

```
END
```

The last line of the program must be an **END** statement, and it is a fatal error if the last line of the program is not an **END** statement. It is also a fatal error for any other line of the program to have an **END** statement. To exit the program early, use the **STOP** statement. See chapter 1 for more information and example programs.

FOR

The **FOR** statement is used for coding *pre-test* loops that use an index numeric variable. A pre-test loop is a loop where the test to determine whether to execute the loop body or exit the loop occurs at the start of the loop, before the loop body. Every **FOR** statement must have a corresponding **NEXT** statement using the same index numeric variable. The **FOR** statement comes in two forms:

```
FOR varname=expression1 TO expression2 STEP expression3
FOR varname=expression1 TO expression2
```

The spaces shown are required. The upper-case words are reserved words, the varname is any valid numeric scalar variable, and the expressions are any valid numeric expressions. The behavior of the second form is identical to the first form except that expression3 is automatically set to the value 1, since the default increment is 1. After the **FOR** statement, zero or more statements may occur, followed by the corresponding **NEXT** statement which has the form:

```
NEXT varname
```

The same varname used as the loop index in the **FOR** statement must be used in the corresponding **NEXT** statement. The **FOR** and **NEXT** statements are a nicer way to write this:

```
   LET limit = expression2
   LET increment = expression3
   LET varname = expression1
L1 IF (varname - limit) * SGN(increment) > 0 THEN L2
      body of loop goes here
   LET varname = varname + increment
   GOTO L1
L2 REM
```

where L1 and L2 are line numbers, and limit and increment are special hidden variables created by the BASIC implementation that are not accessible to the programmer. See chapter 5 for more information and example programs.

GOSUB

The **GOSUB** statement is used to call a subroutine. The general form of the **GOSUB** statement is:

```
GOSUB lineno
```

where lineno is a unsigned integer value specifying the line number of a line that exists in the program. After the **GOSUB** keyword and a space, exactly one existing line number must be specified. See chapter 13 for more information and example programs.

GOTO

The **GOTO** statement is used to branch unconditionally to a new statement. The general form of the **GOTO** statement is:

```
GOTO lineno
```

where lineno is a unsigned integer value specifying the line number of a line that exists in the program. After the **GOTO** keyword and a space, exactly one existing line number must be specified. See chapter 5 for more information and example programs.

IF

The **IF** statement is used to branch conditionally to a new statement. The general form of the **IF** statement is:

```
IF condition THEN lineno
```

where lineno is an unsigned integer line number that exists in the program, and condition follows this pattern:

```
expression relop expression
```

where relop is one of these six relational operators:

```
<  <=  =  >=  >  <>
```

The **IF** keyword is followed by an expression, then a relational operator, then another expressions, then the **THEN** keyword, and finally by a line number. The types of the expressions must match so that both are numeric expressions or both are strings. For strings, only equals and not equals are permitted. At runtime, the expressions are evaluated, then compared using the specified relational operator. If the expression is true, the program will branch to the line number specified. If the expression is false, the program will do nothing and continue with the line immediately following the **IF** statement. The line number specified after the **THEN** keyword must exist in the program. See chapter 7 for more information and example programs.

INPUT

The **INPUT** statement is used to read data into one or more variables from the keyboard. The general form of the **INPUT** statement is:

```
INPUT v1, v2, ...
```

where vN is a scalar numeric variable, a scalar string variable, or a subscripted numeric array variable. The **INPUT** keyword is followed by a space and then one or more comma-delimited variable names. This will prompt the user with a question mark, and then read values from the keyboard. For numeric variables, numeric values must be used. Each input value is separated by a comma. The number and types of value entered must match the number and types of the variables in the list after the **INPUT** keyword. See chapter 3 for more information and example programs.

LET

The **LET** statement is used to assign a value to a variable. The **LET** statement comes in two forms:

```
LET stringvariable = stringliteral
LET numericvariable = numericexpression
```

In either case, the value on the right-hand side of the equals sign is computed and stored in the variable specified between the **LET** keyword and the equals sign. See chapter 3 for more information and example programs.

NEXT

The **NEXT** statement is used to mark the end of the loop body for a **FOR** statement. The general form of the **NEXT** statement is:

```
NEXT varname
```

where varname is a scalar numeric variable name specifying the loop index that was specified in a corresponding preceding **FOR** statement. Every **NEXT** statement must have a corresponding **FOR** statement using the same scalar numeric variable for the index. See the **FOR** statement description for details. See chapter 5 for more information and example programs.

ON

The **ON** statement is used to create a multi-way branch. The general form of the **ON** statement is:

```
ON condition GOTO L1,L2,...
```

where condition is a numeric expression and L1, L2, etc. are unsigned integer line numbers of lines that exist in the program. The **ON** keyword is followed by a numeric expression, then the word **GOTO**, and finally by a list of one or more comma-delimited line numbers. The line numbers must exist in the program. The numeric expression is evaluated at runtime and then converted to an integer. If it is 1, the program branches to the first line number specified. If it is 2, the program branches to the second line number specified, etc. It is an error if the expression evaluates to a number less than one, or a number greater than the number of line numbers in the line number list after the **GOTO** keyword. There must be at least one line number after the **GOTO** keyword. See chapter 8 for more information and an example program.

OPTION

The **OPTION** statement is used to set the minimum array subscript to zero or one. The form of the **OPTION BASE** statement is:

```
OPTION BASE X
```

Where X is either 0 or 1. The digit specifies the minimum array index permitted for the program. The default is zero if no **OPTION** statement exists in the program. This statement must occur before any array access or the **DIM** statement, and it can occur only once. See chapter 10 for more information and example programs.

PRINT

The **PRINT** statement is used to send output to the terminal. There are three general forms of the **PRINT** statement:

```
PRINT
PRINT expression1 delimiter1 ... expressionN
PRINT expression1 delimiter1 ... expressionN delimiterN
```

In the first form, any pending output is output if it exists, and a newline is output. If no pending output exists, then this will output a blank line. In the second form, one or more expressions are output and then a newline is output. In the third form, one or more expressions are output and the final delimiter marks the output as pending so no newline is output in most cases.[2].

Two possible delimiters exist, the comma and the semicolon. There are 5 equally-sized tabular columns of output. By default, output lines have 80 character columns. The 5 tabular columns have 16 characters each. The comma will cause the output to advance to the next tabular column, and the semicolon will advanced to the next character column. In the third form with a trailing delimiter, the data is appended to the output buffer but not printed until the output buffer has a full line of data.

Expressions are either numeric expressions, string literals, scalar string variables, or a special **TAB()** function. For string literals and scalar string variables, the actual string contents are output without any enclosing double quotes. For numeric expressions, the expression is evaluated to generate a number, and then that number is output. Negative numbers have a leading minus sign, and other numbers have a leading space. All numbers have a trailing space. The **TAB(X)** function requires exactly one numeric argument that will be used to indicate the desired character column X for the next output. See chapter 9 for more information and example programs.

RANDOMIZE

The **RANDOMIZE** statement is used to seed the random number generator used by the built-in **RND** function. The general form of the **READ** statement is:

```
RANDOMIZE
```

If **RANDOMIZE** is not used, the random number sequence generated by calls to the built-in **RND** function will always be the same. The reason behind that design feature of the ECMA-55 Minimal BASIC standard is to make testing of programs using the **RND** function repeatable. When the program must generate a different sequence of random numbers every time it is run, the **RANDOMIZE** statement must be used in the program before any call of the **RND** function. See chapter 7 for more information and example programs.

[2]The actual rules are very complex and depend on whether the pending output would exceed the current output line width See the ECMA-55 Minimal BASIC standard for details.

READ

The **READ** statement is used to read data from the in-program data stored in the **DATA** statement(s). The general form of the **READ** statement is:

```
READ v1, v2, ...
```

where vN is a scalar numeric variable, a scalar string variable, or a subscripted numeric array variable. After the **READ** keyword there must be a comma-delimited list of one or more variables. A string variable can hold any value, but a numeric variable can only hold a numeric value from a numeric constant or a string that can be trivially converted to a numeric constant. See chapter 11 for more information and example programs.

REM

The **REM** statement is used to add a comment to the source code of the program. The form of the **REM** statement is:

```
REM UPPER-CASE COMMENT TEXT
```

The space after the **REM** keyword is required unless **REM** is immediately followed by a newline. The text after that space can be any upper-case letters, any digits, a space, single quote, double quote, and any character in this list:

```
():;<>$%_+-*/=^.?!
```

Note that all text in the comment, and in the program, must be valid 7-bit ASCII as shown in table 1 of the ECMA-55 Minimal BASIC standard. See chapter 4 for more information and example programs.

RESTORE

The **RESTORE** statement is used to reset the internal pointer used by the **READ** and **DATA** statements back to the very first data item. The general form of the **RESTORE** statement is:

```
RESTORE
```

See chapter 11 for more information and an example program.

RETURN

The **RETURN** statement is used to exit a subroutine that was entered with **GOSUB** and continue execution on the line immediately following the **GOSUB** that invoked the subroutine. The general form of the **RETURN** statement is:

```
RETURN
```

It is a fatal error to attempt to execute a **RETURN** statement when no corresponding **GOSUB** statement was executed. See chapter 13 for more information and example programs.

STOP

The **STOP** statement will halt execution of the program immediately. The general form of the **STOP** statement is:

```
STOP
```

Unlike the **END** statement, the **STOP** statement may occur multiple times and on any line of the program except the very last line, which must be an **END** statement. See chapter 12 for more information and an example program.

TAB()

The **TAB()** function requires exactly one numeric argument that will be used to indicate the desired character column X for the next output. If the position specified is less than the current position, the current buffered output is displayed, and the output buffer is initialized to a value of X-1 spaces. If the position specified is greater than the maximum display column M, then the formula $X - M \times \lfloor \frac{X-1}{M} \rfloor$ will be used to compute the value actually used. Most implementations use a value of 80 for M. The **TAB()** function can only be used in a **PRINT** statement. The argument can be any valid numeric expression that returns an integer value between 1 and M.[3] See chapter 9 for more information and example programs.

[3]The actual rules for **TAB()** are very complex; see the ECMA-55 Minimal BASIC standard for complete details.

Appendix D

BASIC Numeric Functions

This appendix has a list of all of the numeric functions in ECMA-55 Minimal BASIC. Each entry in the list briefly describes the input argument, if any, and the return value of the function. Input arguments must be surrounded by parenthesis, and no space between the function name and the left parenthesis is permitted.

ABS(X)

The **ABS** function takes exactly one numeric argument, and will return the absolute value of the argument X. In math texts this is written as $|x|$ and the result if X is positive is that X is returned unchanged. If X is negative, then $-X$ is returned, yielding a positive value. The value returned is sometimes called the magnitude of X.

ACOS(X)

The **ACOS** function takes exactly one numeric argument, and will return the principal value of the arc cosine of X. The arc cosine of X is the angle θ whose cosine is X, where $-1 \leq \mathrm{X} \leq 1$. This function returns the principal value θ of the arc cosine of X in radians, where $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$. ***This function is only available when extensions are enabled with the -X option.***

ANGLE(X,Y)

The **ANGLE** function takes exactly two numeric arguments, and will return the angle in radians between the positive x-axis and the vector joining the origin to the point with coordinates (X, Y), where $-\pi <$ **ANGLE(X, Y)** $\leq \pi$. X and Y must not both be zero. Note that counterclockwise is positive. For example, **ANGLE(1,1)** returns $\frac{\pi}{4}$ radians. ***This function is only available when extensions are enabled with the -X option.***

ASIN(X)

The **ASIN** function takes exactly one numeric argument and will return the principal value of the arc sine of X. The arc sine of X is the angle θ whose sine is X, where $-1 \leq \mathrm{X} \leq 1$. This fnction returns the principal value θ of the arc sine of X in radians, where $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$. ***This function is only available when extensions are enabled with the -X option.***

`ATN(X)`

The `ATN` function takes exactly one numeric argument, and will return the principal value of the arc tangent of X. The arg tangent of X is the angle θ whose tangent is X, where $-\infty \leq \mathrm{X} \leq +\infty$. This function returns the principal value θ of the arc tangent of X in radians, where $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$.

`CEIL(X)`

The `CEIL` function takes exactly one numeric argument, and will return the smallest integral value that is not less than X. That is, it always rounds up, so `CEIL(0.5)` is 1, and `CEIL(-0.5)` is 0. In math texts, this function is written as $\lceil x \rceil$ and is called the ceiling function. ***This function is only available when extensions are enabled with the `-X` option.***

`COS(X)`

The `COS` function takes exactly one numeric argument, and will return the cosine of X, where the argument X is in expressed in radians. See chapter 17 for an example program.

`COSH(X)`

The `COSH` function takes exactly one numeric argument and will return the hyperbolic cosine of X. ***This function is only available when extensions are enabled with the `-X` option.***

`CSC(X)`

The `CSC` function takes exactly one numeric argument, and will return the cosecant of X, where the argument X is in expressed in radians. ***This function is only available when extensions are enabled with the `-X` option.***

`COT(X)`

The `COT` function takes exactly one numeric argument, and will return the cotangent of X, where the argument X is in expressed in radians. ***This function is only available when extensions are enabled with the `-X` option.***

`DATE`

The `DATE` function takes no argument, and returns the current date as a number in the form YYDDD, where YY are the last two digits of the year and DDD is the current day of the year, with the first day being 001. For example, on May 3, 2021 the result would have been 21123. ***This function is only available when extensions are enabled with the `-X` option.***

DEG(A)

The **DEG** function takes exactly one numeric argument, an angle A in radians, and will return the angle in degrees. For example, **DEG(PI/2)** returns 90. You can do this with a user-defined function in pure ECMA-55 Minimal BASIC using a function like this:

```
10 DEF FND(A) = A*180.0/3.141592653589793
```

However, that uses up one of your twenty-six user-defined functions, and the function name is not very intuitive. ***This function is only available when extensions are enabled with the -X option.***

EXP(X)

The **EXP** function takes exactly one numeric argument, and will return e^X, where e is the well known constant 2.71828 . . ., the base of natural logarithms.

FP(X)

The **FP** function takes exactly one numeric argument and returns the fractional part. For example, **FP(3.14159)** returns 0.14159. You can do this with a user-defined function in pure ECMA-55 Minimal BASIC using a function like this:

```
10 DEF FNI(X)=X-SGN(X)*INT(ABS(X))
```

However, that uses up one of your twenty-six user-defined functions, and the function name is not very intuitive. ***This function is only available when extensions are enabled with the -X option.***

INT(X)

The **INT** function takes exactly one numeric argument, and will return the largest integer value not greater than the supplied numeric argument. That is, it always rounds down, so **INT(1.4)** returns 1, and **INT(-1.4)** returns -2. In math texts, this function is written as $\lfloor x \rfloor$ and is called the floor function.

IP(X)

The **IP** function takes exactly one numeric argument and returns the integer part. For example, **IP(3.14159)** returns 3. You can do this with a user-defined function in pure ECMA-55 Minimal BASIC using a function like this:

```
10 DEF FNI(X)=SGN(X)*INT(ABS(X))
```

However, that uses up one of your twenty-six user-defined functions, and the function name is not very intuitive. ***This function is only available when extensions are enabled with the -X option.***

`LOG(X)`

The **LOG** function takes exactly one numeric argument, and will return the natural logarithm of X, where $X > 0$. It is a fatal error if $X \leq 0$. In math texts, this function is written as $\ln(x)$.

`LOG10(X)`

The **LOG10** function takes exactly one numeric argument, and will return the base 10 logarithm of X, where $X > 0$. It is a fatal error if $X \leq 0$. In math texts, this function is written as $\log_{10}(x)$. ***This function is only available when extensions are enabled with the -X option.***

`LOG2(X)`

The **LOG2** function takes exactly one numeric argument, and will return the base 2 logarithm of X, where $X > 0$. It is a fatal error if $X \leq 0$. In math texts, this function is written as $\log_2(x)$. ***This function is only available when extensions are enabled with the -X option.***

`MAX(X,Y)`

The **MAX** function takes exactly two numeric arguments, and will return the larger (algebraically) of the two values. For example, **MAX(-3,-2)** returns -2, **MAX(3,2)** returns 3, and just for completeness, **MAX(3,3)** returns 3. ***This function is only available when extensions are enabled with the -X option.***

`MAXNUM`

The **MAXNUM** function takes no argument, and will return the largest normal floating point value which the compiler can support. Unlike most functions, the **MAXNUM** function is essentially a named constant, so it can be used in a **DATA** statement. You can achieve a similar effect with a user-defined function in pure ECMA-55 Minimal BASIC using a function like this:

```
10 DEF FNM=2^1024*(1-2^53)
```

or if you are using using 32bit floats (**-s** option), you would use a function like this:

```
10 DEF FNM=2^128*(1-2^24)
```

However, that uses up one of your twenty-six user-defined functions, the function name is not very intuitive, and it cannot be used in a **DATA** statement. ***This function is only available when extensions are enabled with the -X option.***

`MIN(X,Y)`

The `MIN` function takes exactly two numeric arguments, and will return the smaller (algebraically) of the two values. For example, `MIN(-3,-2)` returns −3, `MIN(3,2)` returns 2, and just for completeness, `MIN(-3,-3)` returns −3.

`MOD(X,Y)`

The `MOD` function takes exactly two numeric arguments, and will return something. The second argument *cannot* be zero. If it is, the program will print an appropriate error message and terminate. For example, `MOD(-5,3)` returns 1, `MOD(-4.4,3.1)` returns 1.8, and `MOD(5,3)` returns 2. The function can be defined mathematically as:

$$\texttt{MOD(X,Y)} = X - Y \times \left\lfloor \frac{X}{Y} \right\rfloor, \quad Y \neq 0$$

This function is only available when extensions are enabled with the `-X` option.

`PI`

The `PI` function takes no argument, and will return the floating point value of π. Unlike most functions, the `PI` function is essentially a named constant, so it can be used in a `DATA` statement. You can achieve a similar effect with a user-defined function in pure ECMA-55 Minimal BASIC using a function like this:

```
10 DEF FNP=3.141592653589793
```

However, that uses up one of your twenty-six user-defined functions, the function name is not very intuitive, and it cannot be used in a `DATA` statement. ***This function is only available when extensions are enabled with the `-X` option.***

`RAD(A)`

The `RAD` function takes exactly one numeric argument, an angle A in degrees, and will return the angle in radians. For example, `RAD(90)` returns 1.5707963267948966, which is approximately $\frac{\pi}{2}$. You can do this with a user-defined function in pure ECMA-55 Minimal BASIC using a function like this:

```
10 DEF FNR(A) = A*3.141592653589793/180.0
```

However, that uses up one of your twenty-six user-defined functions, and the function name is not very intuitive. ***This function is only available when extensions are enabled with the `-X` option.***

`REMAINDER(X,Y)`

The **REMAINDER** function takes exactly two numeric arguments, and will return the remainder of the division of the first argument by the second. The second argument *cannot* be zero. If it is, the program will print an appropriate error message and terminate. For example, `REMAINDER(-5,3)` returns -2, `REMAINDER(-4.4,3.1)` returns -1.3, and `REMAINDER(5,3)` returns 2. The function can be defined mathematically as:

$$\texttt{REMAINDER(X,Y)} = \begin{cases} X - Y \times \left\lfloor \frac{X}{Y} \right\rfloor, & \text{if } Y \neq 0,\ X > 0; \\ 0, & \text{if } Y \neq 0,\ X = 0; \\ X - Y \times \left\lceil \frac{X}{Y} \right\rceil, & \text{if } Y \neq 0,\ X < 0. \end{cases}$$

This function is only available when extensions are enabled with the `-X` option.

`RND`

The **RND** function takes no argument, and will return a pseudo-random floating point value v in the range $0 \leq v < 1$. **Note well:** The sequence used will always be the same unless the **RANDOMIZE** statement is executed before the call to the **RND** function. See chapter 7 for more information and example programs.

`ROUND(X,N)`

The **ROUND** function takes exactly two numeric arguments, and will return the value of X rounded to N decimal digits to the right of the decimal point (or -N digits to the left if $N < 0$). It is equivalent to this ECMA-116 Full BASIC expression:

```
INT(X*10^N+.5)/10^N
```

This function is only available when extensions are enabled with the `-X` option.

`SEC(X)`

The **SEC** function takes exactly one numeric argument, and will return the secant of X, where the argument X is in expressed in radians. ***This function is only available when extensions are enabled with the `-X` option.***

`SGN(X)`

The **SGN** function takes exactly one numeric argument, and will return a value indicating the sign of the argument. If $X < 0$, -1 is returned. If $X > 0$, 1 is returned. Finally, if $X = 0$, 0 is returned. In math texts, this function is written as $\text{sgn}(x)$ and is called the *signum* function.

`SIN(X)`

The `SIN` function takes exactly one numeric argument, and will return the sine of X, where the argument X is expressed in radians. See chapter 17 for an example program.

`SINH(X)`

The `SINH` function takes exactly one numeric argument and will return the hyperbolic sine of X. ***This function is only available when extensions are enabled with the `-X` option.***

`SQR(X)`

The `SQR` function takes exactly one numeric argument, and will return the principal square root of the argument X where $X \geq 0$. The case $X < 0$ is a fatal error. In math texts, this function is written as $\sqrt{x}$ and returns a complex number for $x < 0$. Since Minimal BASIC does not support math with complex numbers, negative values of X are not permitted. The square root of x is the number which when squared will yield x:

$$(\sqrt{x})^2 = x$$

`TAN(X)`

The `TAN` function takes exactly one numeric argument, and will return the tangent of X, where the argument X is expressed in radians. `TAN(X)` is not defined for some values of X. The permitted values of X are those where $X \in \Re \mid X \neq (\pi/2 + k \times \pi)$ for any integer k.

`TANH(X)`

The `TANH` function takes exactly one numeric argument and will return the hyperbolic tangent of X. ***This function is only available when extensions are enabled with the `-X` option.***

`TIME`

The `TIME` function takes no argument, and returns the time elapsed since the previous midnight, expressed in seconds, with the value of `TIME` at midnight being zero. For example, the result of calling `TIME` at 01:02:50 would have been 3770. ***This function is only available when extensions are enabled with the `-X` option.***

TRUNCATE(X,N)

The **TRUNCATE** function takes exactly two numeric arguments, and will return The value of X truncated to N decimal digits to the right of the decimal point (or -N digits to the left if N<0). It is equivalent to this ECMA-116 Full BASIC expression:

```
IP(X*10^N)/10^N
```

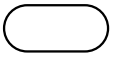
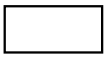
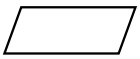
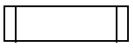
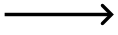
This function is only available when extensions are enabled with the -X option.

Appendix E

Flowcharts

This appendix provides example flowcharts for all common flowchart shapes, together with the ECMA-55 Minimal BASIC code to implement them. Some flowcharts must be modified to allow easy implementation, and in those cases, the modified flowcharts are also shown. The symbols used in this book are shown in Table E.1.

Table E.1: Flowchart Symbols

Symbol Name	Description	Shape
Terminal	These are used to show the start and stop of a program or subroutine.	
Process	Used for assignment statements.	
Input/Output	Used for statements that read input or write output.	
Connector	Used to connect disjoint parts of a flowchart.	
Decision	Evaluates a condition and chooses an output flowline based on the result.	
Predefined Process	Invokes a subroutine.	
Flowline	Connects other symbols. Arrowhead shows direction of logic flow.	
Nested Code	Used for nested code blocks like loop bodies or true and false blocks in if and if/else statements.	

Let's talk about some straight-line code now, as seen in Figure E.1.

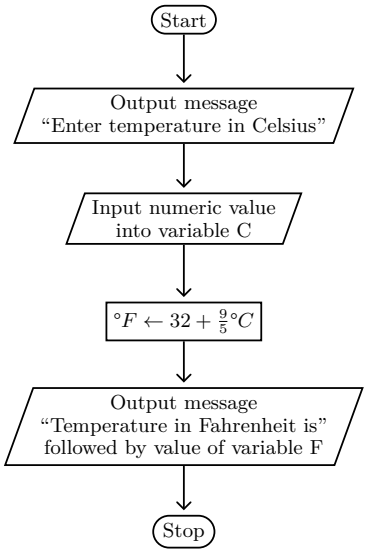


Figure E.1: Algorithm flowchart for converting °C to °F

```
10 REM CELSIUS TO FAHRENHEIT CONVERTER
20 REM
30 PRINT "ENTER TEMPERATURE IN CELSIUS";
40 INPUT C
50 LET F=32+(9/5)*C
60 PRINT "TEMPERATURE IN FAHRENHEIT IS";F
70 END
```

Figure E.2: Minimal BASIC program for converting °C to °F

A flowchart is a special kind of directed graph, with geometric shapes as nodes and lines with arrows on them representing the edges. The flowchart for the algorithm in Figure E.1 is in the left-hand column. The corresponding Minimal BASIC program is in the right-hand column in Figure E.2. Since the algorithm flowchart has no diamonds, it is straight-line code. The Minimal BASIC program is easy to create from the flowchart, with exactly one line of Minimal BASIC corresponding to each flowchart node after the Start.

Notice that the **INPUT** and **PRINT** statements are in parallelograms, and the **LET** statement is in a rectangle. These shapes are known as the *Input/Output* and *Process* symbols, respectively. The beginning and **END** are in rounded rectangles, which are known as *Terminal* symbols. Minimal BASIC programs always begin on their first line, so the Start has no corresponding code in the program. All of the symbols are connected with lines that have arrowheads indicating the direction of logic flow. Those lines with arrowheads are called *Flowlines*.

The flowchart is programming language agnostic. That is, it doesn't depend on or specify the use of any particular programming language. Instead, it represents the algorithm we want to implement. You can use the same flowchart to implement the program in any reasonable procedual, iterative programming language.

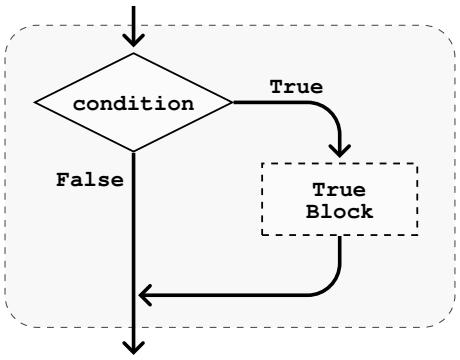


Figure E.3: If flowchart

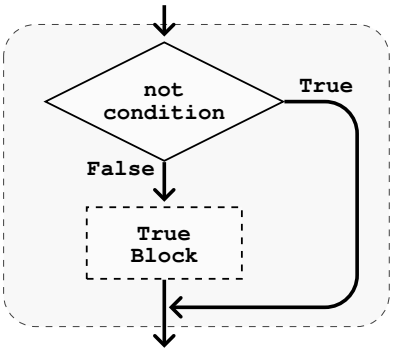


Figure E.4: If flowchart for Minimal BASIC

The flowchart in figure E.3 is re-implemented for Minimal BASIC in figure E.4. Notice that **not** prefixes the original condition in the second flowchart. This technique is sometimes referred to as *flipping the condition*. Instead of running the True block when the condition is true, the logic says skip the True block when the condition is not true. It is effectively the same thing, but the flipped condition works better when you want to implement the program in the Minimal BASIC language.

```
10 REM SIMPLE IF EXAMPLE
20 PRINT "DO YOU WANT THE TRUE BLOCK (Y/N)";
30 INPUT N$
40 IF N$<>"Y" THEN 60
50 PRINT "TRUE BLOCK"
60 END
```

Figure E.5: `EX01.BAS`

```
$./EX01
DO YOU WANT THE TRUE BLOCK (Y/N)? Y
TRUE BLOCK
$./EX01
DO YOU WANT THE TRUE BLOCK (Y/N)? N
$
```

Figure E.6: `EX01.BAS` runtime output

A Minimal BASIC version of the algorithm is implemented in Figure E.5, and the runtime output is shown in Figure E.6. The True Block is one line, line 50. The not condition is on line 40, and has the program jump over the True Block to line 60 if **`N$`**'s value is not "**`Y`**".

The diamond is known as the *Decision* symbol. Decision symbols must have at least two output Flowlines, but no other symbol can have multiple output Flowlines. Start must not have any input Flowline. All other symbols can have multiple input Flowlines.

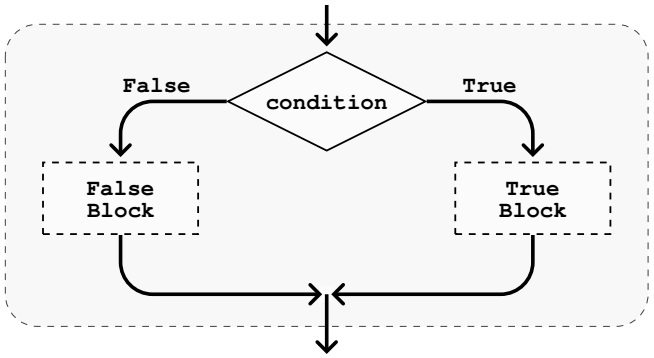


Figure E.7: If with Else flowchart

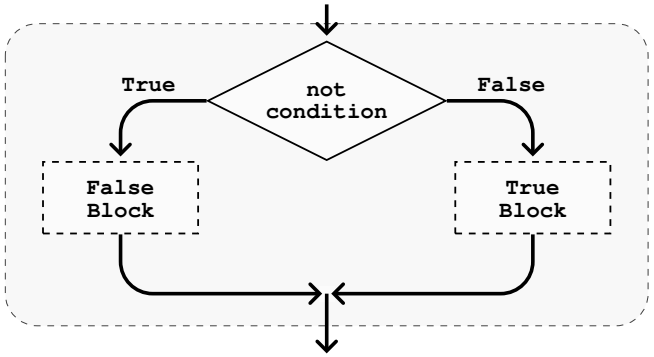


Figure E.8: If with Else flowchart for Minimal BASIC

The flowchart in figure E.7 is re-implemented for Minimal BASIC in figure E.8. A Minimal BASIC version of the algorithm is implemented in Figure E.9, and the runtime output is shown in Figure E.10 on page 208.

```
10 REM IF X==4 WITH ELSE EXAMPLE
20 PRINT "ENTER X";
30 INPUT X
40 IF X <> 4 THEN 70
50 PRINT "TRUE BLOCK"
60 GOTO 80
70 PRINT "FALSE BLOCK"
80 END
```

Figure E.9: `EX02.BAS`

```
$./EX02
ENTER X? 3
FALSE BLOCK
$./EX02
ENTER X? 4
TRUE BLOCK
$
```

Figure E.10: `EX02.BAS` runtime output

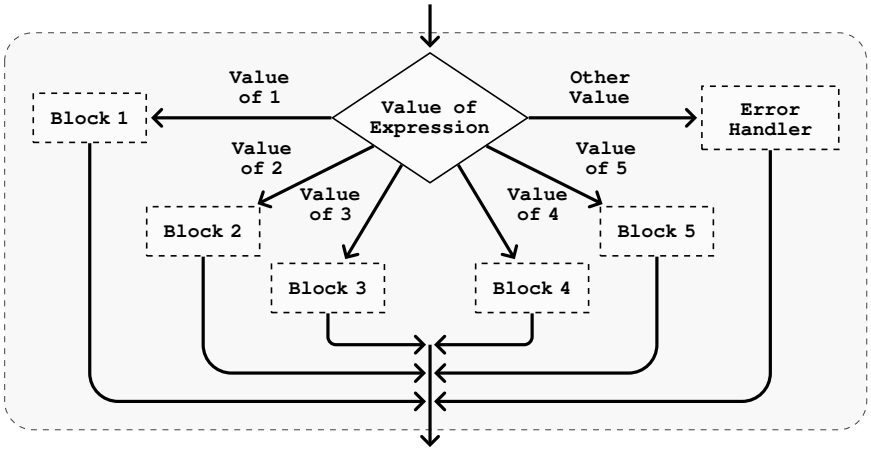


Figure E.11: Five-way branch flowchart

A Minimal BASIC version of the algorithm shown in Figure E.11 is implemented in Figure E.12, and the runtime output is shown in Figure E.13.

```
10 REM MULTI-WAY BRANCH EXAMPLE
20 PRINT "ENTER VALUE";
30 INPUT V
40 IF V > 0 THEN 70
50 PRINT "ERROR HANDLER"
60 GOTO 190
70 IF V< 6 THEN 90
80 GOTO 50
90 ON V GOTO 100, 120, 140, 160, 180
100 PRINT "BLOCK 1"
110 GOTO 190
120 PRINT "BLOCK 2"
130 GOTO 190
140 PRINT "BLOCK 3"
150 GOTO 190
160 PRINT "BLOCK 4"
170 GOTO 190
180 PRINT "BLOCK 5"
190 END
```

Figure E.12: `EX03.BAS`

```
$./EX03
ENTER VALUE? 1
BLOCK 1
$./EX03
ENTER VALUE? 2
BLOCK 2
$./EX03
ENTER VALUE? 3
BLOCK 3
$./EX03
ENTER VALUE? 4
BLOCK 4
$./EX03
ENTER VALUE? 5
BLOCK 5
$./EX03
ENTER VALUE? 6
ERROR HANDLER
$./EX03
ENTER VALUE? 0
ERROR HANDLER
$
```

Figure E.13: `EX03.BAS` runtime output

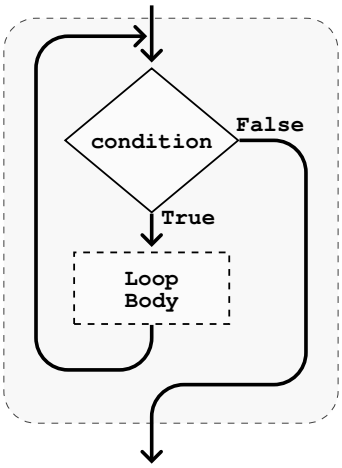


Figure E.14: While Loop

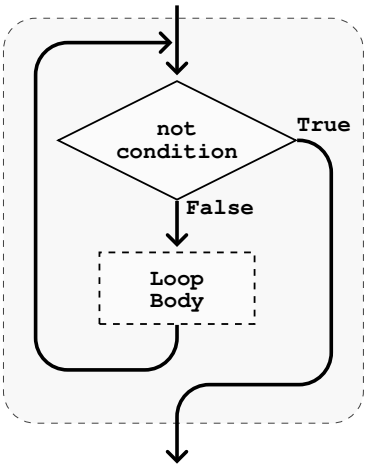


Figure E.15: While Loop for Minimal BASIC

The flowchart in figure E.14 is re-implemented for Minimal BASIC in figure E.15. The Minimal BASIC version of the algorithm is shown in figure E.16 and is implemented with a forward jump over the loop body using a flipped condition. If the condition is false, the flipped condition is true and the loop terminates. If the condition is true, the flipped condition is false and the loop body is executed. This is exactly the behavior of a While loop. The runtime output is shown in figure E.17 on page 211.

```
10 REM WHILE X <= 10 LOOP EXAMPLE
20 LET X=1
30 IF X > 10 THEN 90
40 REM BEGIN LOOP BODY
50 PRINT "LOOP ITERATION";X
60 LET X=X+1
70 REM END LOOP BODY
80 GOTO 30
90 PRINT "LOOP DONE"
100 END
```

Figure E.16: `EX04.BAS`

In the program example, the forward jump is implemented with the **`IF`** on line 30. The loop body spans lines 40 through 70, and the **`GOTO`** on line 80 jumps back to line 30 which has the **`IF`** with the loop condition test.

```
$./EX04
LOOP ITERATION 1
LOOP ITERATION 2
LOOP ITERATION 3
LOOP ITERATION 4
LOOP ITERATION 5
LOOP ITERATION 6
LOOP ITERATION 7
LOOP ITERATION 8
LOOP ITERATION 9
LOOP ITERATION 10
LOOP DONE
$
```

Figure E.17: `EX04.BAS` runtime output

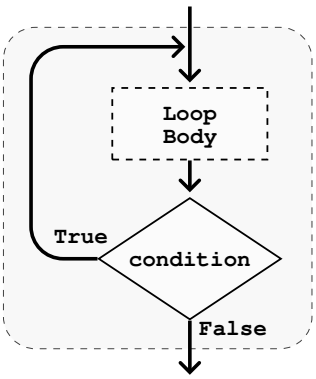


Figure E.18: Do .. While loop

A Minimal BASIC version of the algorithm shown in Figure E.18 is implemented in Figure E.19, and the runtime output is shown in Figure E.20.

```
10 REM DO .. WHILE I <= 10 EIAMPLE
20 LET I=1
30 REM BEGIN LOOP BODY
40 PRINT "LOOP ITERATION";I,"I=";I
50 LET I=I+1
60 REM END LOOP BODY
70 IF I<=10 THEN 30
80 PRINT "LOOP DONE"
90 END
```

Figure E.19: `EX05.bas`

```
$./EX05
LOOP ITERATION 1              I= 1
LOOP ITERATION 2              I= 2
LOOP ITERATION 3              I= 3
LOOP ITERATION 4              I= 4
LOOP ITERATION 5              I= 5
LOOP ITERATION 6              I= 6
LOOP ITERATION 7              I= 7
LOOP ITERATION 8              I= 8
LOOP ITERATION 9              I= 9
LOOP ITERATION 10             I= 10
LOOP DONE
$
```

Figure E.20: `EX05.BAS` runtime output

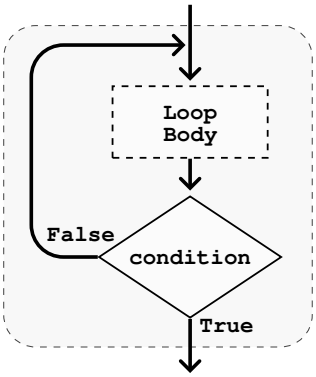


Figure E.21: Repeat .. Until Loop

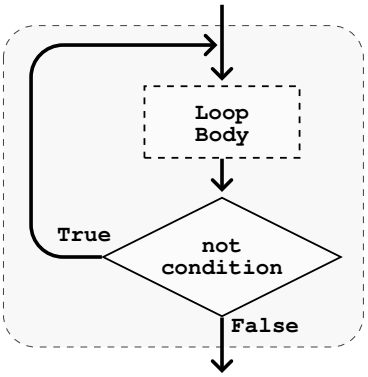


Figure E.22: Repeat .. Until Loop for Minimal BASIC

The flowchart in figure E.21 is re-implemented for Minimal BASIC in figure E.22. You can see the condition in the diamond was flipped and the Minimal BASIC code uses a backward jump with an **IF** statement to run the loop body if the condition is false, which is exactly what a Repeat .. Until loop should do. The Minimal BASIC version of the algorithm is implemented in Figure E.23, and the corresponding runtime output is shown in Figure E.24.

```
10 REM REPEAT .. UNTIL I = 10 EXAMPLE
20 LET I=0
30 REM BEGIN LOOP BODY
40 LET I=I+1
50 PRINT "LOOP ITERATION";I,"I=";I
60 REM END LOOP BODY
70 IF I<>10 THEN 30
80 PRINT "LOOP DONE"
90 END
```

Figure E.23: `EX06.BAS`

```
$./EX06
LOOP ITERATION 1              I= 1
LOOP ITERATION 2              I= 2
LOOP ITERATION 3              I= 3
LOOP ITERATION 4              I= 4
LOOP ITERATION 5              I= 5
LOOP ITERATION 6              I= 6
LOOP ITERATION 7              I= 7
LOOP ITERATION 8              I= 8
LOOP ITERATION 9              I= 9
LOOP ITERATION 10             I= 10
LOOP DONE
$
```

Figure E.24: `EX06.BAS` runtime output

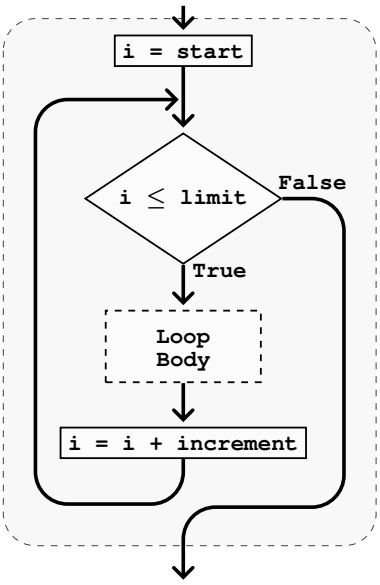


Figure E.25: Ascending Arithmetic For Loop

A Minimal BASIC version of the algorithm shown in Figure E.25 is implemented in Figure E.26, and the runtime output is shown in Figure E.27.

```
10 REM ASCENDING ARITHMETIC FOR LOOP DEMO
20 FOR I=1 TO 10 STEP 1
30 PRINT "ITERATION";I,"I=";I
40 NEXT I
50 PRINT "LOOP DONE"
60 END
```

Figure E.26: `EX07.BAS`

```
$./EX07
ITERATION 1    I= 1
ITERATION 2    I= 2
ITERATION 3    I= 3
ITERATION 4    I= 4
ITERATION 5    I= 5
ITERATION 6    I= 6
ITERATION 7    I= 7
ITERATION 8    I= 8
ITERATION 9    I= 9
ITERATION 10   I= 10
LOOP DONE
$
```

Figure E.27: `EX07.BAS` runtime output

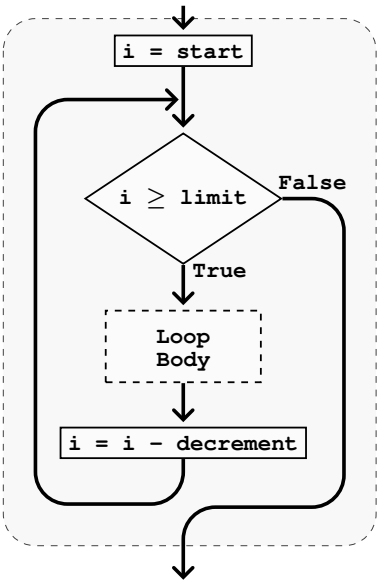


Figure E.28: Descending Arithmetic For Loop

A Minimal BASIC version of the algorithm shown in Figure E.28 is implemented in Figure E.29, and the runtime output is shown in Figure E.30.

```
10 REM DESCENDING ARITHMETIC FOR LOOP DEMO
20 FOR I=10 TO 1 STEP -1
30 PRINT "ITERATION";11-I,"I=";I
40 NEXT I
50 PRINT "LOOP DONE"
60 END
```

Figure E.29: `EX08.BAS`

```
$./EX08
ITERATION 1    I= 10
ITERATION 2    I= 9
ITERATION 3    I= 8
ITERATION 4    I= 7
ITERATION 5    I= 6
ITERATION 6    I= 5
ITERATION 7    I= 4
ITERATION 8    I= 3
ITERATION 9    I= 2
ITERATION 10   I= 1
LOOP DONE
$
```

Figure E.30: `EX08.BAS` runtime output

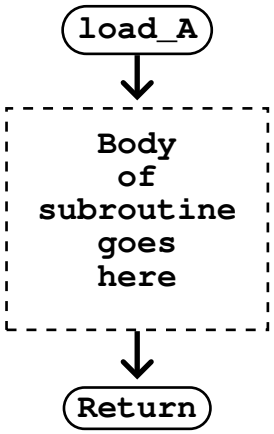


Figure E.31: Subroutine

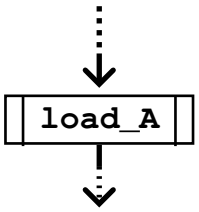


Figure E.32: Subroutine Call

A Minimal BASIC program with a subroutine as shown in Figure E.31 and a call as shown in Figure E.32 is implemented in Figure E.33, and the runtime output is shown in Figure E.34.

```
10 REM SUBROUTINE DEMO
20 FOR I=1 TO 10 STEP 1
30 REM NEXT LINE IS SUBROUTINE CALL
40 GOSUB 80
50 NEXT I
60 PRINT "LOOP DONE"
70 STOP
80 REM SUBROUTINE
90 PRINT "ITERATION";I,"I=";I
100 RETURN
110 END
```

Figure E.33: `EX09.BAS`

```
$./EX09
ITERATION 1    I= 1
ITERATION 2    I= 2
ITERATION 3    I= 3
ITERATION 4    I= 4
ITERATION 5    I= 5
ITERATION 6    I= 6
ITERATION 7    I= 7
ITERATION 8    I= 8
ITERATION 9    I= 9
ITERATION 10   I= 10
LOOP DONE
$
```

Figure E.34: `EX09.BAS` runtime output

The subroutine example spans lines 80 through 100. It is called on line 40 inside the loop body of the ascending arithmetic for loop. Each time the program reaches the `GOSUB` on line 40, it will remember the next line is 50, then it will jump to line 80 and run until reaching the `RETURN` on line 100, at which point it will jump to line 50 and continue the loop.

Appendix F

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a

publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

ABS(), 197
absolute value, 197
ACOS(), 197
algorithm, 2, 9
    append_node, 166
    average, 80
    bubble sort, 113
    conversion
      Celsius to Fahrenheit, 9
      Fahrenheit to Celsius, 11
    delete_node, 171
    dump_raw_storage, 154
    factorial, 37
    Fibonacci sequence, 43
    find_node, 159
    initialize_storage, 157
    load_storage, 158
    maximum, 83
    print_nodes, 156
    search
      binary, 121
      sequential, 99
    Taylor series
      cosine, 47
    update_price, 161
    update_qty, 160
allocate, 150
angle, 197
ANGLE(), 197
append
    singly linked list, 166
append_node, 166
arctangent, 198
arc cosine, 197
arc sine, 197, 198, 203
array, 79
    DIM, 80
    dimension, 133
    element, 79
    index, 79
    integer-indexed, 79
    matrix, 133
    OPTION BASE, 83
    size, 80
    subscript, 79
    vector, 79, 133
arrow, *see* flowchart
ASCII, 2
ASIN(), 197
assignment, 2
    LET, 17
    rectangle, 2
ATN(), 198
average, 80

bas55, 181
binary search, 121
block, 150
branch, 5
    conditional, 22
      diamond, 2
      FOR, 31
      IF, 28
    jump target, 31
    multi-way, 57, 58
      using **IF**, 58
      using **ON**... **GOTO**, 60

unconditional, 31
GOSUB, 107
GOTO, 31
bubble sort, 113

call, 107
CEIL(), 198
ceiling, 198, 199, 201
Celsius, 9
connector, *see* flowchart, *see* flowchart → symbol → connector
constant, 18
COS(), 145, 198
COSH(), 198
cosine, 198, 202
COT(), 198
CSC(), 198

DATA, 91, *see* **READ**, **RESTORE**, 187
DATE, 198
deallocate, 150
debug, 13
decision, *see* flowchart → symbol → decision
DEF, 145, 188
DEG(), 199
delete
singly linked list, 171
`delete_node`, 171
diamond, *see* flowchart
Dijkstra, Edsger W., xi
DIM, 80, 188
dimension, 133
double, 16
`dump_raw_storage`, 154
duplicate key, 162
dynamic memory manager, 150

ecma55, 185
extensions with **-X**
ACOS(), 197
ANGLE(), 197
ASIN(), 197
CEIL(), 198
COSH(), 198
COT(), 198
CSC(), 198
DATE, 198
DEG(), 199
FP(), 199
IP(), 199
LOG10(), 200
LOG2(), 200
MAX(), 200
MAXNUM, 200
MIN(), 201
MOD(), 201
PI, 201
RAD(), 201
REMAINDER(), 202
ROUND(), 202
SEC(), 202
SINH(), 203
TANH(), 203
TIME, 203
TRUNCATE(), 204
element, 79
END, 1, 11, 189
EXP(), 199
exponential function, 199

factorial, 37
Fahrenheit, 9
Fibonacci number, 43
field, 135
`find_node`, 159
flipping the condition, 207
floating point, 16
floor, 52, 199
floor function, 199, *see* **INT**
flowchart, 2, 205
arrow, 2
connector, 72
diamond, 2
parallelogram, 2

predefined process symbol, 107
rectangle, 2
symbol, 205
connector, 205
decision, 205
flowline, 205
input/output, 205
predefined process, 205
process, 205
terminal, 205
terminal symbol, 2
flowline, *see* flowchart → symbol → flowline
FOR, 31, *see* **NEXT**, **STEP**, **TO**, 189
FP(), 199
free list, 150
function
built-in math
ABS(), 197
ACOS(), 197
ANGLE(), 197
ASIN(), 197
ATN(), 198
CEIL(), 198
COS(), 198
COSH(), 198
COT(), 198
CSC(), 198
DATE, 198
DEG(), 199
EXP(), 199
FP(), 199
INT(), 52, 199
IP(), 199
LOG(), 200
LOG10(), 200
LOG2(), 200
MAX(), 200
MAXNUM, 200
MIN(), 201
MOD(), 201
PI, 201
RAD(), 201
REMAINDER(), 202
RND, 202
ROUND(), 202
SEC(), 202
SGN(), 202
SIN(), 203
SINH(), 203
SQR(), 203
TAN(), 203
TANH(), 203
TIME, 203
TRUNCATE(), 204
built-in text
TAB(), 193, 195
user-defined, 145

GOSUB, 107, 190
GOTO, 31, 190

hard-coded, 66, 94
head pointer, 149
heap, 150
initialize, 157
load, 158

IF, 3, 28, 190
index, *see* **FOR**, *see* loop, *see* subscript
`initialize_storage`, 157
INPUT, 9, 11, 191
input/output, *see* flowchart → symbol → input/output
INT(), 52, 199
integer-indexed array, 79
integer part, 199
invoke, *see* call
IP(), 199
iteration, 24, *see* **STEP**

jump target, 31

Kemeny, John George, 1
key, 159

primary, 159
keyword, 1
Kurtz, Thomas Eugene, 1

LET, 3, 11, 191
line
number, 1
width, 2
line number, 1
linked list, *see* singly linked list
list
in **DATA** statements, 94
singly linked, 149
literal value, *see* constant
`load_storage`, 158
LOG(), 200
LOG10(), 200
LOG2(), 200
logarithm function, 200–202, 204
loop, 24
FOR, 189
index, 79
index variable, 31
iteration, 24
post-test, 24
pre-test, 31, 189

matrix, 133
MAX(), 200
maximum, 83
MAXNUM, 200
memory
block, 150
memory leak, 151
MIN(), 201
MOD(), 201
multi-way branch, 57, 58
using **IF**, 58
using **ON**...**GOTO**, 60

NEXT, 31, *see* **FOR**, **STEP**, **TO**, 192
next pointer, 149
node, 149
null pointer, 149

ON...**GOTO**, 60, 192
OPTION, 192
OPTION BASE, 83

parallelogram, *see* flowchart
PI, 201
pointer, 150
porting, 111
post-test loop, 24
pre-test loop, 31
predefined process, *see* flowchart → symbol → predefined process
predefined process symbol, *see* flowchart
primary key, 159
PRINT, 3, 11, 192
comma delimiter, 63
semicolon delimiter, 42
TAB(), 193, 195
`print_nodes`, 156
process, *see* flowchart → symbol → process
pseudo-random number, 52

RAD(), 201
RANDOMIZE, 52, *see* **RND**, 193
random number, *see* pseudo-random number
READ, 91, *see* **DATA**, **RESTORE**, 194
real number, 16
record, 134
rectangle, *see* flowchart
REM, 1, 3, 194
REMAINDER(), 202
RESTORE, 92, *see* **DATA**, **READ**, 194
RETURN, 107, 194
RND, 51, *see* **RANDOMIZE**, 202
ROUND(), 202

scalar, 16
search
binary, 121
sequential, 99

singly linked list, 159
SEC(), 202
sentinel, 94
sequences, 21
sequential search, 99
SGN(), 202
signum function, 202
SIN(), 145, 203
sine, 203
singly linked list, 149
append, 166
delete, 171
search, 159
stored in a matrix, 153
traverse, 156
update, 160, 161
SINH(), 203
sort, *see* bubble sort
SQR(), 203
square root, 203
STEP, 32, *see* **FOR**, **NEXT**, **TO**
STOP, 104, 195
straight-line, 5
string, 16
subroutine, 107
call, 107
GOSUB, 107
RETURN, 107
subscript, *see* array
symbol, *see* flowchart → symbol

TAB(), 193, 195
TAN(), 203
tangent, 203
TANH(), 203
Taylor series, 47, 50
temperature, 9
terminal, *see* flowchart → symbol → terminal
terminal symbol, *see* flowchart
THEN, 28, *see* **IF**
TIME, 203
TO, 32, *see* **FOR**, **NEXT**, **STEP**
traverse
singly linked list, 156
TRUNCATE(), 204

update
singly linked list, 160, 161
`update_price`, 161
`update_qty`, 160
user-defined function, 145, *see* **DEF**

variable, 15
vector, 79