

Programming

**Charles R. Coombs
2914 S. Finley Drive
Midwest City, Oklahoma 73130
April 25, 1992**

Charles R. Coombs

Mr. Coombs has been programming computers for over twenty years. He has written or maintained hundreds of programs in a dozen languages, from machine language on the earliest posting-machine computers to the most sophisticated data manipulation and query languages on the largest mainframes.

His experience extends to traditional data processing applications in retailing and insurance. He has also programmed and maintained sophisticated, distributed applications in defense installations. As a technical analyst for two major computer vendors, he has performed effectively as a system and data administrator working with a number of hardware and operating systems.

Mr Coombs has made his greatest contribution as a user programmer making computer and information technology available to unsophisticated users. His emphasis has always been on the customer and the customer's needs, and how professional information resource managers can most easily meet them.

Table of Contents

Introduction	INTRO-1
Chapter I: The Big Idea	I-1
Chapter II: Syntax	II-1
BASIC Language Elements	II-1
BASIC Instructions	II-1
BASIC Functions	II-5
Reserved Words	II-6
BASIC Commands	II-6
Syntax Notation	II-6
A Subset of BASIC	II-8
Commands	II-8
Instructions	II-11
Functions	II-18
Chapter III: Grammar for Writing Programs	III-1
Simple Structures	III-2
Simple Sequence	III-3
Simple Decision	III-4
Simple Iteration	III-4
Programming Simple Structures	III-6
Flowcharts	III-6
Psuedocode	III-10
Putting It Together	III-11
Complex Structures	III-13
Complex (Nested) IFs	III-14
The CASE Structure	III-16
Complex (Nested) Loops	III-17
Issues Concerning Complex Structures	III-18
Compound Structures	III-18
Compound IFs	III-19
A Compound IF with Nesting	III-25
Compound Loops	III-25

Issues Concerning Compound Structures	III-26
Chapter IV: Grammar for Maintenance	IV-1
Analyzing Structures	IV-2
Identifying IFs	IV-3
Identifying Loops	IV-8
Normalizing Structures	IV-8
Input Loops with Branchbacks	IV-8
Multiple Loop Exits	IV-11
Spaghetti Code	IV-12
The Compleat Grammarian	IV-14
Chapter V: Composition	V-1
The Parthenon Form of the Essay	V-1
The Parthenon Form of the Program	V-4
Building Modules	V-4
Building Programs	V-6
Analyzing Composition	V-8
Chapter VI: Style	VI-1
Grammatical Style	VI-1
Flowcharting Style	VI-1
Psuedocoding Style	VI-3
Compositional Style	VI-4
Style and Optimization	VI-6
Eclat	VI-8
Appendix A: The Example Program	A-1
Problem Description	A-1
Input Specification	A-1
Output Specification	A-3
Control File Input	A-6
Functional Hierarchy	A-8
Program Flowchart	A-9
First Example Listing	A-12
Second Example Listing	A-16
Sample Input	A-20

Sample Output	A-22
Appendix B: Notes on Assembler Language and COBOL	B-1
The IBM 360/370 Assembler Language	B-1
COBOL	B-4

List of Illustrations

Figure III-1: The Simple Structures	III-3
Figure III-2: The Three Simple Loops	III-8
Figure III-3: The Simple IF	III-9
Figure III-4: A Complex (Nested) IF	III-15
Figure III-5: The CASE Structure	III-16
Figure III-6: A Complex (Nested) Loop Structure	III-17
Figure III-7: Compound IFs	III-19
Figure III-8: Three Clauses ANDed Together	III-21
Figure III-9: Parentheses in Compound Structures	III-21
Figure III-10: A Factored and Expanded Compound IF	III-23
Figure III-11: A Compound/Complex IF	III-24
Figure III-12: Conundra	III-26
Figure III-13: The Conundra Decoded	III-27
Figure III-14: Repeating Code Along Adjacent Control Paths	III-28
Figure III-15: Forcing Adjacent Control Paths	III-29
Figure IV-1: Flowchart for the Input Loop Example	IV-7
Figure IV-2: The Multiple Loop Exit Equivalent	IV-11
Figure IV-3: The Spaghetti Code Equivalent	IV-13
Figure V-1: The Parthenon Form of the Essay	V-2
Figure V-2: A Functional Hierarchy	V-6
Figure V-3: Flowchart of the Overlapping Loop Example	V-11
Figure V-4: Functional Hierarchy for the Overlapping Loop	V-12
Figure VI-1: Using a Quadrille Pad for Flowcharting	VI-2

Introduction

Whether you write computer programs for a living or not, you can profitably use this book. If you are a professional programmer, you can create more complex and more reliable work using the ideas in this book than you could without them. If you're a maintenance programmer, you can use the same techniques to modify other peoples' programs as successfully and profitably as you create your own.

If you use computers, but aren't directly paid to program them, you can still use the ideas of this book to make your computer more profitable. These techniques work as well for, say, LOTUS or WordPerfect macros, or for dBASE applications as they do for FORTRAN or for IBM assembler language programs.

This book is about the language of instruction sets. It analyzes the syntax, grammar, composition, and style of a computer program in the Beginners All-purpose Symbolic Instruction Code (BASIC), the most common and readily available of all computer languages. But the important concepts of this book are language- and machine-independent. They work equally well with any programming language running on any computer.

This approach should prove most useful to people who have written or maintained computer programs, but you can fully understand and use it if you've never turned a computer on. If you have an IBM PC, you can run the example program without modification. If you have another computer that supports BASIC, you can do the examples and run the example program with little modification, which should only serve as good practice anyway. If you don't have a computer at all, you can master every important point of the book with pencil and paper alone.

All of these ideas have been tested on hundreds of real-world programs in dozen languages in commercial data processing. There's nothing abstract or academic about any of them.

What this book is not is a treatise on "structured programming." Structured programming began well, promising to spark a revolution that really would have made things better than they were. As most revolutions do, though, this one grew corrupt almost the minute its cadres assumed power.

This because they had a lot of power. Data processing was in a fairly uniform state of disarray in the late 1970s. Backlogs of work that couldn't command an overloaded programmer's attention grew from months to years. Information users weren't getting what they wanted, and heaven help the business whose requirements changed before a program could be written and tested. Structured programming promised a workable alternative to all that.

The promise looked good enough to give structuralist consultants the run of most data processing shops. Unfortunately, most of them didn't actually write programs. They wrote books, rather, about programming. They didn't work directly with programmers. They worked with managers, many of whom were in management precisely because they couldn't program computers. They piled standard on standard without ever making it clear to the people who had to comply with them what the standards were supposed to accomplish.

They created new programming languages intended to merge the grammar of program writing with its syntax, instilling "discipline" into programmers, whom they openly considered lazy and stupid. They wrote with overt scorn of "unstructured" programs, as if a program that runs reliably, producing a specified output from a specified input could exist

Programming

without structure. They derided “unstructured” programmers as though unbelievers were made of silicon putty, rather than flesh, blood, and brains.

Withal, they “structured” everything in sight. The movement came to include structured analysis, structured management, structured database applications — even structured English, presumably for creating structured documentation. It didn’t take long for the movement to run out of control and credibility. Programmers found ways around the standards and continued to program the way they always had.

The revolution fizzled, having become just another political and social fad in American business. By losing sight of the thing they originally set out to do, the structuralists lost everything. Including a lot of programmers who, weary of the constant insults and abuse, got out of programming altogether.

The effect of this is not all bad. Good programmers are becoming harder to find and their services command an ever-higher rate. On the other hand, the backlog is larger than it’s ever been. New applications, which might be written to saturate an Intel i486, don’t challenge the 80286, because vendors don’t believe they can find people to program such difficult applications. There’s even a school that holds we can’t build an application like the Strategic Defense Initiative (“Star Wars”) because programmers can’t handle the complexity of such a project.

Yet, most people can program well enough to do all these things if they can separate the important elements of program writing from the daunting load of baggage the structuralists have loaded onto it. This book aims to help you do that, providing a useful frame of reference in which to work creatively.

Programming: The Method

What this book emphatically doesn't try to do, then, is teach you how to think. It isn't a criticism of the way you program now. It doesn't require that every line of code written before 1987 be rewritten to conform to some structuralist notion of correctness or purity. It doesn't sell copies of this or that self-structuring compiler. Most programmers greatly enjoy what they're doing, and they're quite good at it. My aim in writing this is to make programming more fun even than it already is, and to deepen the sense of wonder most of us feel at the notion of actually being paid for it.

So let us begin.

Chapter I: The Big Idea

Programming languages are languages: They follow the same rules that any spoken language does, although they're considerably simpler than any spoken language. A program is a piece of writing that a computer can understand.

Four attributes, therefore, describe every program, just as they describe any other piece of writing. These are

- Syntax
- Grammar
- Composition
- Style.

Syntax deals with the structure of clauses as well as with the meaning of words. In computer languages, clauses are built out of operators (instructions, commands) and operands (objects). The syntax of a programming language is thus the way that combinations of operators and operands cause a computer to behave. A program is syntactically correct if it runs and if it produces a specified output from a specified input.

Grammar deals with the structure of sentences, that is, with combinations of clauses. If a program runs, and if it produces a desired output from a given input, then the program is grammatically correct. This is to say that grammatical rules describe sequences of operations that the computer can understand.

Grammars fall into two major categories: There are prescriptive grammars, like the latinate grammars of the 19th and early 20th centuries that generations of American English teachers inflicted on their students, and there are descriptive grammars that seek to examine

existing written work to determine its structure. The grammar in this book is descriptive, and is just as useful for analyzing the operation of existing programs as it is for writing new ones.

Composition deals with the structure of paragraphs and of written documents, including computer programs. It refers to the arrangement of grammatical structures (sentences, paragraphs) within a piece of writing. If a program runs, and if it produces a desired output from a given input, then the program is properly composed. Composition, like grammar, describes. It doesn't prescribe.

Style deals with syntactical, grammatical, and compositional choices a writer or programmer makes in the course of writing or programming. Style deals with "good" and "bad" elements of writing. It is never anything but prescriptive. A "good" style helps you program efficiently and reliably. A "good" style makes your work easy for another human being to read and maintain. In a world in which 80% of all programming work is maintenance work, a good style gets you money and clout.

For purposes of this book, a computer "program" is a piece of writing on which a computer operates to produce a specified output from a specified input. This means that to be a program, it has to run without ending "abnormally" as defined by the computer system it runs on, and it has to run predictably. The computer has to understand it, therefore, even if no living human does, or it just isn't a program at least for our purpose.

A program is a good program to the extent that people comprehend it as readily as a machine does. It's quite possible to write good programs. Indeed, it's easier to write good programs than bad ones: Every programmer is, after all, a human being.

Chapter II: Syntax

If you've ever written a BASIC program, you should skip this chapter. If you've never written one, scan it, then use it for reference if something in the examples is syntactically unclear to you. There are dozens of good texts on BASIC syntax at your local library if you want to study it in more detail.

It's easier to write for a computer than for another human being. Computer languages subsume only a tiny subset of spoken languages. The only tense is present. The only mood is imperative. The predominate parts of speech are verbs, nouns, and conjunctions.

Appendix A contains a working example of a program. There are two code examples, but they're syntactically equivalent. You should refer to the sample program throughout your study of this chapter.

BASIC Language Elements

BASIC language elements include instructions that modify and present data in applications, functions that given a value return another value, and commands that manage the BASIC interpreter or compiler itself. Remember that BASIC, like every other computer language, is an application program. Its function is to write other programs.

BASIC Instructions

BASIC instructions are also called "statements." Every BASIC instruction consists in three parts, as does any instruction in nearly every computer language.

An "operator" is the action you want the computer to take. It's possible to write the most complex program with seven or eight BASIC operators, although later implementations of the language make many more operators available. Operators are the verbs of BASIC.

“Operands” are items of information on which you want the computer to operate. These are the nouns of BASIC. Some of them can be modified adjectivally, the most common adjectives being ordinal numbers.

“Comments” are items of information intended for human use. They have no effect on the operation of the computer.

Most implementations of BASIC also require that instructions have unique, ascending statement numbers. Although probably just as many don’t require this, the most common implementations do. The examples in this book thus all make use of statement numbers.

The following is an example of a BASIC instruction:

```
1000 INPUT "PLEASE TYPE YOUR NAME ",N$ ' GET THE NAME
```

The operator is the word “INPUT.” It tells the computer to place the words “PLEASE TYPE YOUR NAME” on the screen, then to wait until you type in some character data enclosed in double quotation marks, terminating the string with the ENTER key.

The operands are the string “PLEASE TYPE YOUR NAME ” and the name of a location in storage, N\$, at which you wish to store the typed-in information.

The text following the single quote (') is a comment. You have to place the quote at least one space beyond the end of the last operand, but the comment itself can occur at any position beyond the quote, including the space immediately following it. You can also use the REM instruction to turn an entire line into a comment.

One of the big advantages BASIC often has over many other languages is that you can run BASIC instructions interactively, immediately executing instructions you enter at the computer screen. This isn’t true of every BASIC implementation; however, it is true of

almost every BASIC written for small computers, including all the variants of the IBM PC. You should make extensive use of this convenient feature throughout your study of the language.

In fact, you can try this now. Start BASIC on your computer (usually with BASIC, BASICA, or GWBASIC). If you change your mind and wish to get back out of BASIC, type SYSTEM. Assuming you want to continue with this, type in the command as shown, but without the line number. Omitting the line number tells the BASIC interpreter that you want the instruction executed immediately. An instruction entered this way is called a “direct mode” instruction.

When you hit ENTER, assuming you entered the command correctly, you should see the words in quotes appear on the screen. Type in a few characters — not more than 256 — enclosed in double quotes, and hit ENTER again. The cursor will move down a line.

Now type the instruction

PRINT N\$

and hit ENTER. You should see the character string you typed.

If you type in something that doesn’t make sense to the computer, you get back a message that says “SYNTAX ERROR.” Later, you’ll see that the computer can detect grammatical and compositional errors as well. There is no such thing as an error in style.

Operands are the input data for instructions. Usually, operands are *variables*. The term “variable” refers to the name of a location in storage, *i. e.*, the name of an address. In general, what instructions do is take data from one location in storage, do something to it, then store

the result at another location. Variables give the programmer a convenient means of keeping track of these data storage locations.

Various BASIC implementations set different rules for defining variables. Commodore BASIC, for example, requires that variables begin with a letter, and that they be no longer than two characters, plus one character denoting the variable type. Microsoft's GWBASIC lets you define variable names of any length, but recognizes only the first 40 characters. As do other BASICs, GWBASIC allows you to add a special character to define the variable's type.

The word "type" refers to the kind of information stored at the location named by a variable. You can refer to your manual for a more detailed discussion of data types, but the concept should become clear through the example program.

Briefly, the computer sees all data, whether character strings, floating point numbers, or integers, as a string of ones and zeros. For example, the letter "C," a character string, is "01100111" to the computer. So is the integer 67. BASIC can translate what undifferentiated strings of ones and zeros represent.

You, the programmer, are responsible for letting BASIC know what binary sequences like 01100111 are supposed to mean. You do this by the way you define the variable name. If you end the name with a dollar sign (\$), then you've told BASIC that 01100111 means the letter "C." If you end the name with a percent (%) sign, you've told BASIC to view it as the integer 67. If you don't put anything after the name, or if you use a bang (!), you've defined it as a single-precision, floating point number, most useful for calculations. If you really want a lot of accuracy in calculation, you can end your variable name with a pound sign (#) to make it a double-precision floating point number.

If you're truly curious about what floating point numbers look like, you can study them in your manual. You don't really have to understand this representation to program effectively in BASIC, except to know that you use them in calculations that put many digits to the right of the decimal point, or that produce very large or very small numbers.

At several points in the example program of Appendix A, you have to use an instruction that expects data in a different form than you read or generate it in. Look at the way the example program uses the VAL function to convert a numeric character string into a value that the computer can use for computation.

BASIC Functions

Functions are really built-in microprograms that allow you to perform easily those operations, such as computing an inverse hyperbolic cosecant, that the writers of BASIC interpreters believe ought to occur most frequently. You can write your own functions if you like.

The inputs to functions are called "arguments." The function itself acts like an extended instruction. The result of executing the function is to change the variable argument to the function, storing the result at an address named by another variable.

An example of a function is

COS (Y)

which computes the cosine of the angle, in radians, stored at the location named Y. You generally use functions as operands for instructions, as in

```
1000 LET X=COS(Y)
```

which places the cosine of the single-precision, floating-point number (in radians) at Y into the location named X.

Reserved Words

BASIC gets confused when you define variable names in a way that makes it impossible to tell whether you want that word to refer to a variable name or to some other BASIC entity. Examples are AND, OR, IF, FN (or any word that begins with FN), and such BASIC commands as STOP, LIST, and SAVE.

Your manual provides a list of these words, called “reserved words,” but you’ll get a syntax error if you try to define a variable with one. It isn’t worth the effort of a detailed study to memorize what these words are.

BASIC Commands

Commands are instructions to the BASIC compiler as an application program. LIST, which causes BASIC to print your a program in memory to the screen, is a BASIC command. STORE, which causes BASIC to write a program in memory on an external device like a diskette, is another. As a rule, your programs don’t issue BASIC commands, although they can. You use commands as an aid to writing programs.

Syntax Notation

The word “expression” refers to the form of a BASIC operand. You can express a character string as a quoted string, for example, or you can express it as the output of a function, or you can express it indirectly as the location in the computer’s memory at which it resides.

You use syntax notation to specify a syntactically correct expression. The notation in this book is almost universally used to describe programming expressions in every computer language.

These are the rules for specifying a BASIC expression

- Surround a required entity with braces ({}).
- Surround an optional entity with square brackets ([]).
- Surround information you enter yourself (file names, for example) in angle brackets (<>).
- Separate choices with the pipe symbol (|).
- Indicate repeating entries with ellipses (. . .).
- If you have to type something exactly as it is in your specification, write it in upper-case letters.
- If you decide what to type in an instruction, use a word that symbolizes the information, and write that word in lower-case letters.

Here's an example of this notation as it applies to a common BASIC command:

```
SAVE <filespec> [{,A | ,P}]
```

The word "SAVE" is in upper-case letters because it has to be entered just as it is. The symbol "filespec" is in lower-case letters because it refers to the qualified file name you want to give your file on some magnetic medium. You type the actual name there, not the word "filespec."

You can terminate this particular command at the file specification. For that reason, the following specification is in brackets. If you do put something there, however, then you must use either "A" or "P." That's the meaning of the braces enclosed within the brackets.

It's important to understand this notation, not only because you need it to learn BASIC instructions and commands, but because if you write functions, you can use this notation to show every programmer who uses a function of yours how to invoke it. If you call programs with parameter lists, you can easily and accurately document the way a user should start your program. This saves a lot of time and pain in documentation, and makes it possible for a

greater number of people to use your work. That's good. You cannot otherwise become rich and famous as a programmer.

A Subset of BASIC

The following instructions, commands, and functions are sufficient to write much more complicated programs than those in this text. If you want to know more, or if you want to use graphic functions, data entry screens, and suchlike in BASIC, refer to your manual. There are too many variations in BASIC implementations of these instructions to get deeply into them here.

As you review this section, you should have BASIC running on your computer. Test these definitions until you're sure you know what they do. Realize, too, that if you're in the middle of entering a BASIC program, you can check the validity of anything you're programming by typing in the command in question without a file number, pressing ENTER, and using PRINT to evaluate the result. Once you've reassured yourself, go ahead and enter the instruction with a line number. When you save your program, only the numbered statements go to the disk.

Commands

The first command everyone wants to know is SYSTEM. This gets you out of BASIC and back to the operating system. You can code this command in a BASIC program when you want to return to the operating system upon program completion.

Here are some equally useful commands:

CONT

When you're running a BASIC program, you can stop it by typing in CTRL-BRK (Hold down the CTRL key and press the BREAK key) or CTRL-C. You can also stop it with a STOP or END statement in the program. The CONT command resumes execution at the

point at which you stop it. You can also use a GOTO instruction entered in direct (no line number) mode to resume execution of a stopped program at the line number that is the operand of the GOTO instruction.

CONT is sometimes useful in debugging. Be aware, though, that if you edit the program in any way, this command won't work. Since your purpose in stopping the program is normally going to be precisely to allow you to edit the thing, you probably won't use this command much. I don't.

DELETE [<line number>][-<line number>]

This allows you to erase lines of code you've entered into memory. Deleted lines don't get executed or written to the disk when you SAVE your program.

FILES

This command shows you a list of files from the disk directory you were in when you invoked BASIC. Useful when you want to load something and you can't remember its name.

LIST [<line number>][-<line number>][<device>]

This command prints lines from your program. If you don't enter any operands, it lists the entire program to the screen. If you enter one line number, the command lists that line only on the screen. If you enter a line number and a dash, then every line after the first line number gets printed to the screen. If you enter a line number, a dash, then a higher line number, the range of numbers from the low number to the high number prints to the screen.

If you enter a printer port name, such as LPT1:, as the <device>, the listing goes to whatever device is connected to that port. Usually, that's a printer, making the LIST command the most common way of printing BASIC programs.

LOAD <FILESPEC>[,R]

This command gets a program stored on a disk into memory. The ,R parameter isn't supported in every implementation of BASIC. When it is, it runs your program after loading it.

NEW

This clears out the program in memory, making the memory available for a new program. You only need to do this if you intend to type the new program in. LOAD replaces any code you might have in memory with the loaded program.

RUN [<line number>]

This executes the program in memory, beginning at <line number>. If you omit the operand, execution begins at the lowest line number.

Some BASICs implement a second form of this command. This is

RUN <filespec>[,R]

which loads the program from disk before running it. If you code the ,R parameter, then any data files you had open before you loaded the new program remain open during its execution. The new program replaces the old in either case. This is useful if your program grows to be so big that you have to use “overlays” to run the whole thing in the memory you have. In a day and age when memory is the cheapest thing you can buy for a computer, I doubt you'll use this form of the RUN command much.

SAVE <filespec>[{ ,A | ,P}]

This command writes a program from memory to disk. If you specify nothing after <filespec>, then BASIC saves the file as a “tokenized,” ready-to-run file. Specifying ,A causes BASIC to store the program as an ASCII file that you can work on with your word processor or editor. Specifying ,P lets you store the file in “protected” mode, which keeps it

from being listed or edited once it's been loaded. As a security measure, storing a program with the ,P option was thoroughly defeated years ago.

In the IBM PC world, if you don't put an extension on <filespec>, BASIC automatically appends .BAS to the name you do enter. This is true even if you use the ,A option.

I use the ,A option as a matter of preference. The simplest word processor makes a better text editor than the best BASIC interpreter. I use the interpreter for making minor edits during debugging, then SAVE the program both with and without the ,A option. Just be sure your statement numbers are in ascending order: If you load a BASIC program with two identical statement numbers, the second statement overlays the first. If you try to load a statement without a number, you get an "illegal direct mode statement" error. If you load statements that aren't numbered in ascending order, BASIC sorts them into ascending order for you. The effect of this can be jarring.

Instructions

These instructions constitute a small subset of the instructions available in most implementations of BASIC. Use your manual to study additional instructions available to you. Meantime, these give you enough BASIC syntax to clarify everything in this book, the best part of which is language independent anyway.

```
CLOSE [[#]<file number>[, [#]<file number . . .>]
```

This instruction complements the OPEN instruction, explained below. If you don't code this instruction, most BASICs close any open files for you upon reaching an END or a SYSTEM instruction.

DATA <constant list>

You use this instruction to define a list of constants for use with a READ statement. The most common use of this is to store a table of information with your BASIC program, using the READ statement to make the information available to your program. Here's an example.

Say you think your bank has made a mistake on your statement. You want to enter the five checks returned to with your statement, add them up, and compare the list to the list on your statement. In real life, you'd use a spreadsheet for this, or you'd do it by hand. Real life isn't what we're dealing with just now, though, and the following program would give you what you want, more or less:

```
1000 TOTAMT=0
1010 FOR I=1 TO 5
1020 READ CKDATE$,WHAT$,CKAMOUNT
1030 TOTAMT=TOTAMT+CKAMOUNT
1040 PRINT CKDATE$,WHAT$,CKAMOUNT,TOTAMT
1050 NEXT I
1060 PRINT
1070 PRINT TOTAMT
1080 END
1090 DATA "03/24/92","SM's Hall of Exotic Terpsichore",520.00
1100 DATA "03/31/92","Rupert the Bookie",1525.79
1110 DATA "04/01/92","Moral Rearmament",250.00
1120 DATA "04/09/92","Felicia's Pleasure Palace",155.00
1130 DATA "04/24/92","Citizens for Decency",2549.32
```

The READ statement inputs the information in the DATA statements three values (two character and one floating point decimal) at a time. You could have used one DATA statement to hold all eighteen constants. READ simply gets as many values as you tell it to, one after the other, until it runs out of information.

DIM <subscripted variable list>

This instruction reserves storage in your program for a subscripted array. You can think of an array as a table like the one in DATA statements of the previous example. The example program of Appendix A uses such an array to name the fields in a logical input file record.

END

This stops your program's execution, closes all your files (which we'll discuss later), and returns you to the command level screen you invoked when you started BASIC. You have to include either END or SYSTEM as the last instruction you execute in a BASIC program. If you use SYSTEM as an instruction this way, you'll be returned to the command interpreter, *e. g.* COMMAND.COM in the MS/DOS operating system. Play around with these instructions.

```
FOR <variable> = <x> to <y> [STEP <z>]
    .
    .
    .
NEXT [<variable>][,<variable>. . .]
```

The example given with the DATA statement contains a FOR . . . NEXT loop. In that example, I stores the number of times you've been through the loop. It begins at one. When your program executes the NEXT I statement, NEXT adds one to I, then checks whether the value of I now exceeds five. If it does, NEXT passes control to the statement following itself. If it doesn't, NEXT passes control back to the FOR statement.

STEP gives you a lot of control over the way you use FOR . . . NEXT loops. You can increment the loop counter I by two, for example, by coding

```
FOR I=1 TO 10 STEP 2
```

The most common use of STEP, though, is to count backwards through a loop by making STEP negative, for example

```
FOR I=10 to 1 STEP -1
```

The example program uses this kind of loop to take blanks from the ends of character strings.

```
GOSUB <line number>
      .
      .
      .
RETURN [<line number>]
```

This instruction allows you to execute a block of code out of sequence. The code block is called a *subroutine*. You code the GOSUB in the main body of your program. You normally put the subroutine after the END statement, coding RETURN at its end. GOSUB transfers control to the first line number of the subroutine. The RETURN statement transfers control to the first statement following the GOSUB statement.

```
GOTO <line number>
```

The GOTO is an unconditional transfer of control. The example program we'll discuss in the later chapters of this book makes liberal use of GOTO to implement grammatical structures like IF-THEN-ELSE-ENDIF and DO WHILE.

```
IF <expression>
  THEN {<statement> . . . | <line number>}
[ELSE {<statement> . . . | <line number>}]
```

IF provides a conditional transfer of control. "Expression" refers to the statement of some condition like $\text{SIN}(X) > 0.5$ or $\text{VRBL\$} = \text{"ABCDE"}$ or $A = B$. If the condition is true, the statement that receives control is the code immediately following THEN, or the line number coded after THEN. The grammatical methods in this book don't use ELSE: It's easier and more effective to code that structure using GOTOs.

```
INPUT[;] [<"prompt string">;]<list of variables>
```

The <"prompt string">, which is shown that way because it has to be in double quotes if you code it, is a string variable you want displayed on the screen. The first example in this section used an INPUT statement with a prompt in this way.

Executing an INPUT statement with the semicolon after the prompt string displays the prompt string and a question mark. The INPUT statement without a prompt string displays only a question mark. Either way, the program is asking you to type something directly from the keyboard into the variable(s) in the second operand. If there's more than one such variable, separate them by commas. You let the program know you're finished entering data by pressing the ENTER key.

If you use a semicolon after the INPUT instruction itself, pressing ENTER won't put a carriage return/line feed sequence at the end of the input data. The cursor will remain at the end of the data you typed.

INPUT is most useful in interactive coding of a type we don't do much of in this book.

It is helpful in studying BASIC syntax. Here's an example:

```
1000 K%=0
1010 INPUT "DON'T JUST SIT THERE - TYPE SOMETHING ";STUFF$
1020 PRINT STUFF$," IS THING NUMBER ",K%," I'VE DONE FOR YOU."
1030 K%=K%+1
1040 IF STUFF$="XXXXX" THEN 1060
1050 GOTO 1010
1060 END
```

When you get tired of this program, which shouldn't take so very long, type in five Xs to make it stop.

If you use a comma after <prompt string> instead of the semicolon, INPUT won't put the question mark on the screen.

INPUT #<file number>,<variable list>

This instruction gets information into a computer's memory from a disk or tape device. You specify the file number in an OPEN statement, which we'll deal with anon. INPUT # works like the READ statement, filling in the variable list from records on your sequential device.

Unlike the variables in the DATA statement, the variables on a disk or tape have to be separated into groups that exactly correspond to the variables in <variable list>. These groups are separated by a carriage return and line feed combination, and are called “records.” The stored values themselves are commonly called “fields.”

A variant of this statement is LINE INPUT #. That instruction gets up to 255 characters into a single string variable. The instruction appears in the example program.

[LET] <variable> = <expression>

This instruction is so important to BASIC programming that it’s usually coded only implicitly, (A\$=B\$ vs LET A\$=B\$, for example). It replaces the value of a variable with some other value. This can be a string in quotes, a number, a mathematical expression (A+B+D, for example), a concatenated string (A\$+B\$), another variable, and so forth. The example program abounds in these statements.

**OPEN {<filespec>|<device>}
[FOR <mode>] AS [#]<file number>
[LEN=<record length>]**

Different BASICs implement this command differently. Consult your manual. This OPEN statement works for most PC-based BASIC implementations. It makes a file on a disk or diskette available to your program. You actually access the data via an INPUT # instruction.

OPEN tells the BASIC interpreter which file on the disk you want your program to use. It also tells BASIC how you want to use the file: You can create it, add to it, or simply read it. The file number you give it is the number you refer to in subsequent INPUT # and PRINT # instructions.

The following OPEN instruction creates a file. If you have a file with the same file specification, this instruction erases and replaces it with the new file:

```
OPEN "C:\USR\CONTROL.FIL" FOR OUTPUT AS #2
```

The following instruction opens an existing file to read it. The file must exist, and BASIC won't let you issue an instruction that would write to it:

```
OPEN "C:\USR\CONTROL.FIL" FOR INPUT AS #1
```

You can write to a file without destroying it if you open it thus:

```
OPEN "C:\USR\CONTROL.FIL" FOR APPEND AS #3
```

A PRINT # instruction adds its record to the end of the existing file.

These OPEN instructions deal only with *sequential* files; that is, files whose records (repeating groups of like information) have to be accessed one at a time. To get to record 95 in a sequential file, you generally have to read records one through 94.

BASIC offers another kind of file, called a *random access* file. You can access an individual record in this file, regardless of where it's physically stored without having to read the records that precede it. Almost all of the programs one writes in BASIC use sequential files. You don't need random files to master any of the techniques in this book, and you can easily learn them from your user's manual should you run up against a need for one. We don't, therefore, work with random files here.

```
PRINT [<list of expressions>]
```

This instruction writes information to the screen. It's very useful for debugging: When your program ends badly, it returns you to the BASIC interpreter/editor without resetting any of your variables or closing any of your files. If your problem stems from bad information in a variable, you can PRINT that variable in direct mode to see what's stored at that location. You can also issue INPUT # statements in direct mode to replace the values of the input

variables with the values of fields from the disk. You can PRINT the input variables to see what your stored disk records look like.

The example program makes use of the PRINT instruction to indicate that it's alive during execution. If it stops writing to the screen, you know the program has died and you probably need to reboot or turn off the computer to regain access to it. Most of the techniques in this book specifically seek to keep this kind of thing from happening, but people make mistakes.

```
PRINT #<file number>,[USING <string expression>]  
    <list of expressions>
```

The file number is the one you used in the OPEN instruction. You can't read from or write to a file that isn't open. USING is an option that allows you to format data for presentation in reports. You can look up the ways you use this option, but we don't need it here. The list of expressions is usually a group of variables (they can be constants) that you want to save on your diskette as a record.

Functions

There are many functions in BASIC. The ones you need to understand the examples of this text are

CHR\$(n)

returns the ASCII character representation of the single character n, which is a decimal number in the range of 0 to 255. Remember the binary value 01100111? If you've asked BASIC to read think of this value as the integer 67, you can't turn around and tell BASIC to do something that regards it as the letter C. Letting, say, X\$=CHR\$(67) defines another variable that does represent the letter C. You can also use CHR\$ as an alternative representation of certain special characters, like the comma or semicolon, that are used in BASIC expressions and might be confusing when used directly as data. The example

program can use CHR\$(9), instead of the Ventura tab <9>, for example, which is the TAB character for a great many word processors and desktop publishers.

EOF[<file number>]

returns a “TRUE” or “FALSE” every time you issue an INPUT # instruction against the file whose file number is the argument of EOF. If you get a TRUE, then the INPUT # has accessed the last record in the file. Issuing another INPUT # instruction after that causes an abnormal termination.

The first instructions in a loop that gets data, one record at a time, then processes it are usually of the form

```
100 IF EOF(1) THEN 999
110 INPUT #1,A$,B$,C,D%,E
```

The actual loop processing follows the instruction of line 110. When that instruction reads the last line, it sets EOF(1) to TRUE. Statement 100 is the first instruction in the loop. When the loop control structure returns control to line 100 with EOF(1) true, control transfers to the first instruction following the loop. The example program has at least two of these loops.

LEFT\$(<string> ,n)

BASIC is probably most famous for its ability to handle character strings. This instruction allows you to specify an integer *n* between 1 and 255. The LEFT\$ function returns the leftmost *I* characters of <string>. The character string <string>, by the way, has to contain more characters than *n*.

LEN(<string>)

This function returns the length in characters of its character string argument. The example program uses this as a loop counter for trimming blanks off the ends of character strings.

MID\$(<string> , n , [, <m>])

This function returns a string <m> characters long beginning at position <n> of <string>. The example program teems with this function.

RIGHT\$(<string> , <n>)

In most BASICs, this function returns the rightmost <n> characters of <string>. I've seen one BASIC compiler that returned the leftmost characters of the string counting <n> positions from the right. Go figure.

All implementations of BASIC for small computers that I've encountered make RIGHT\$ the mirror image of LEFT\$, which is as it should be. The example program, which is a PC program, assumes that this is the case.

SPACE\$(X)

returns a string of X spaces. This is useful because it keeps you from having to code “ ” when you want 30 spaces to, say, clear a string variable with.

STR\$(<n>)

provides a string value of the numeric expression <n>. This is how you convert numeric data to string data so you can take, for example, the rightmost nine digits.

CHR\$ also converts numeric to string data; however, CHR\$ takes as its argument only a single byte of information — that's what an integer between 0 and 255 is. STR\$ converts any number BASIC can deal with.

VAL(<string>)

If you can convert a number to its character string representation, it stands to reason you should be able to convert a string of numeric characters (0 through 9) to a number. Sure enough, that's what VAL does.

There are literally scads of other commands, instructions, and functions in most BASIC implementations. These give you enough BASIC syntax to understand the discussion of grammar, composition, and style as they apply to the example program. You can also write your own functions if BASIC for any reason falls short of what you want in this area. The syntax of a language is always the easiest thing to master. When you master the simple ideas in this book, you'll be an ace in any language.

Chapter III: Grammar for Writing Programs

Grammar describes the structure of sentences. Grammars — there are many — fall into two classes: descriptive and prescriptive.

Prescriptive grammars tell you how a sentence ought to be structured. They're necessary in certain contexts, because people, rightly or wrongly, judge you on the way you speak and write. Grammarians who tell you not to say “ain't,” “literally scads,” or “between he and I,” therefore, really have your best interest at heart. The same people might tell you you'll do better as a programmer if you wear pin-stripes and woollens than if you wear jeans and a T-shirt. They're right, and you should listen to them.

Programming language grammars, the chief fruits of the structuralist movement, are uniformly prescriptive, probably for the same reasons. Data processing managers, though, rarely judge you by the way you structure programs. They judge you, rather, by the speed and, sometimes, the reliability with which you solve problems. Most users, who ultimately have the most to say concerning your success as a programmer, never even see your programs. They couldn't care less whether you write the most spasmodic spaghetti code as long as they get the timely information they need to make money.

Prescriptive grammars are therefore inappropriate for programming, however worthwhile they may prove socially when applied to human languages.

Descriptive grammars, on the other hand, assume that if you understand it, it's grammatical. They seek to describe the structure of comprehensible sentences without making any assertion concerning what it should be. Since we've held throughout that a

program is grammatical if it runs from beginning to end, consistently producing a specified output from a specified input, and since the overwhelming mass of existing code doesn't *even* fulfil the demands of current structuralist grammars, what programmers need more than anything else is a grammar they can apply to existing programs.

All this says is that if the computer understands it, we should try to learn why the computer understands, then describe that process to a human being. No one who makes any difference to you cares how nicely you state this understanding, as long as you can prove you have it.

Because you control the process, it's easier to describe gramatically a program that doesn't already exist. Once you've written a few such programs, you can easily adapt a descriptive grammar to the analysis of existing work.

The example program of Appendix A contains two listings. The first, YAFVPTP, is a complete program written according to the rules of this chapter. You may wish to refer to it as you continue.

Simple Structures

Most programmers refer to computer-language sentences as *structures*. "Normal" structures are a small subset of all the possible structures you can build in a computer language. A normal structure conforms to a norm: That is, it's agreed-on, usual, standard, customary, canonical, ordinary, common, and generally recognized — all these terms mean the same thing. If you speak in terms of normal structures, you can converse intelligibly with anyone who's ever written a computer program.

There are three normal structures: simple sequence, selection, and iteration. Figure III-1 represents the normal structures pictorially.

Simple Sequence

In a simple sequence, instructions follow one another in a straight line. “Control,” defined as the next instruction in the execution sequence, proceeds from one instruction to the next in line. Beyond providing the easiest way to understand control, simple sequences aren’t grammatically interesting.

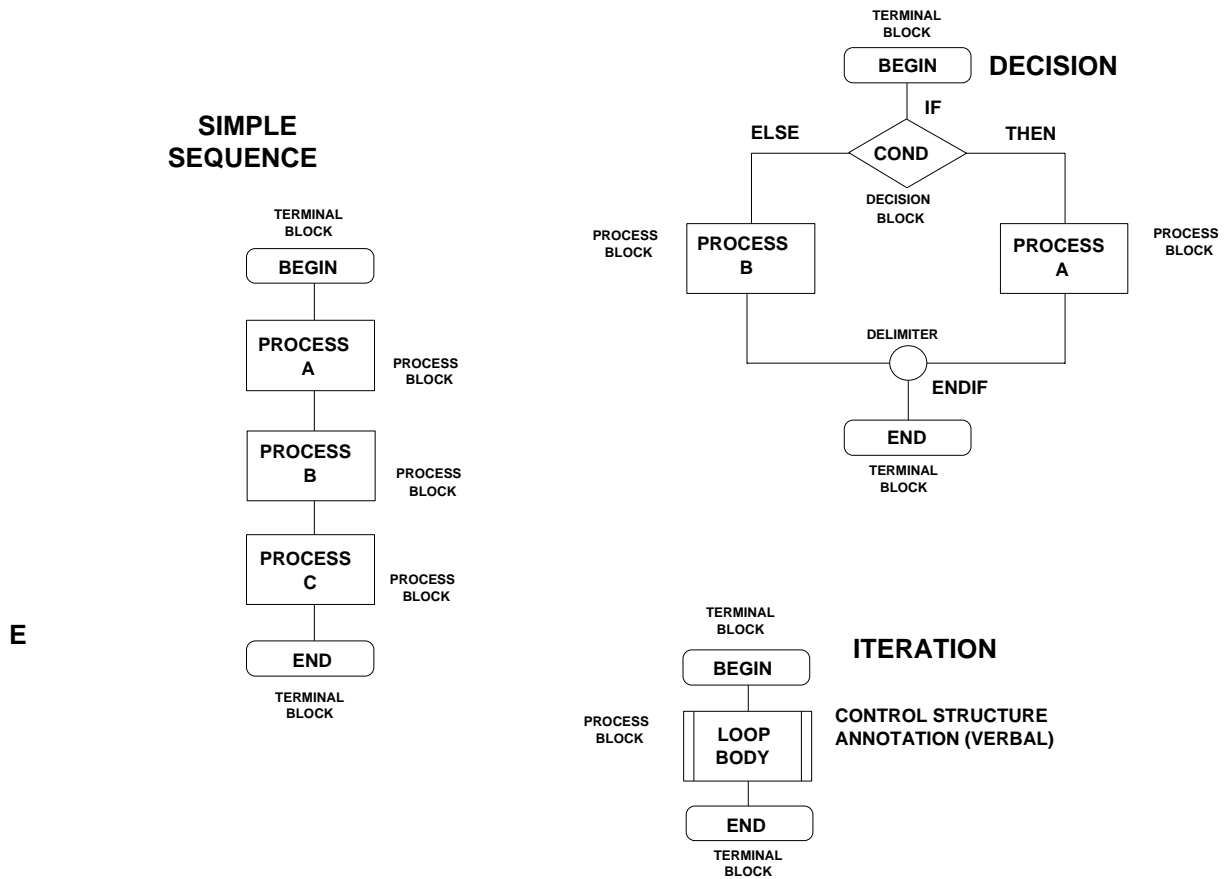


Figure III-1: The Simple Structures

In the picture, instructions (or sequences of instructions) appear as “process blocks.” The blocks contain some kind of description of what the program’s instructions are supposed to do.

The rounded–corner boxes at the top and bottom are “terminal blocks.” They show the beginning and end of any structure or collection of structures you represent this way.

The line connecting the boxes from terminal to terminal represents control.

Simple Decision

Computers make binary choices. They decide between “yes” and “no,” between “true” and “false,” between “zero” and “one,” for example.

You represent simple decisions with “decision blocks.” Control flows into the decision block from the top, and emerges from the left and right sides. Your program executes the code on the right or left side of the decision block depending on a binary condition that you write in the decision block itself. The right side is “TRUE,” “THEN,” “YES,” “ONE,” and such. The left side is, respectively, “FALSE,” “ELSE,” “NO,” “ZERO,” and so forth.

The circle under the bottom point of the decision block is a “delimiter.” This limits the *scope* of the structure so that you can determine when you’re back to executing instructions that control would pass to even if the decision weren’t there. The “scope” of any decision, then, is everything between a decision block and its delimiter.

Simple Iteration

Iterations are always called “loops,” because control passes through them more than once. Every loop has three associated substructures: These are

- The *initialization* structure
- The *control* structure, and

- The *body* or *execution* structure.

The initialization structure of a loop is usually a simple sequence, but it can be any structure. It can occur anywhere in a program, as long as it executes before the loop does, but typically it immediately precedes the loop it initializes. It sets conditions for the first iteration of the loop.

The control structure is a decision to return control to the first instruction of a loop, or to send control to the instruction following the loop. Control structures come in three normal flavors: REPEAT WHILE, REPEAT UNTIL, and REPEAT FOR. Some programmers use the word “DO” instead of “REPEAT.” The terms are interchangeable here.

A simple loop, as opposed to a “compound” or “complex” loop, which we’ll discuss later, has only one control structure. That control structure is always a simple decision.

A REPEAT WHILE control structure checks the condition which if true continues the loop, passing control to the beginning of the loop. If the condition is false, control passes to the first instruction after the loop, terminating it. Prescriptive grammarians often specify that REPEAT WHILE precede the loop body. That isn’t grammatically necessary, and REPEAT WHILE can occur either at the top or the bottom of a loop.

A REPEAT UNTIL control structure checks the condition which if true causes control to transfer to the first instruction following the loop. If the condition is false, control returns to the first instruction of the loop. Like the REPEAT WHILE, this control structure can occur either before or after the loop body.

A REPEAT FOR control structure causes the body code to execute a fixed number of times. It can be at the top (before the body) or bottom (after the body) of a loop, but a number of languages implement REPEAT FOR through specialized instructions, like the

FOR . . . NEXT loop in BASIC or the “Branch on Count” in IBM assembler language. These instructions usually come with a syntactical requirement to locate them at the beginning or the end of the loop body. There’s no grammatical requirement to use these special instructions for coding REPEAT FOR structures, but they’re handy. Do what makes your boat float.

To build a normal loop, you do have to locate the control structure either at the beginning or the end of the loop, and you have to separate control structures to the extent possible from the loop body code. Some languages make it necessary to share the body function with the control function, but even these never make it necessary for you to share more than one instruction between the two functions.

The same holds for initialization and control structures. The FOR part of the FOR . . . NEXT loop, for example, shares initialization and control functions, but otherwise the substructures remain separate.

Programming Simple Structures

This book describes grammatical constructs in two ways. One is the flowchart, the other is psuedocode. You write a program structure by drawing a flowchart (assuming the structure is complicated enough to warrant it), then by writing psuedocode in the comment portion of your program statements that exactly describes the program code itself. There is power in this.

Flowcharts

Flowcharts correspond to sentence diagrams in English. Sentence diagramming is a relatively tedious exercise in spoken languages, and most people avoid it after leaving high school. For this reason, flowcharting has fallen into disuse in most programming shops. Flowcharting is extremely useful, however, for programming or analyzing complicated

programs, especially if you don't try to use it for documenting programs. We'll make liberal use of it here.

For flowcharting instruction sets, you need only half a dozen symbols, not counting the line that symbolizes control. These are

- The process block, a rectangle
- The decision block, a diamond
- The CASE block, a circular section
- The terminator, a box with rounded corners
- The delimiter, a small circle, and
- The barred process block, a rectangle with vertical lines on either side.

We've already discussed what all but two of these symbols mean. We'll discuss the CASE block later.

The barred process block indicates only that the process it describes is a loop body or that it appears in more detail in another flowchart. If the process is a loop body, then the control structure appears in words beside the block. Note that the barred box has nothing to do with "subroutines," the great number of programmers who use it this way notwithstanding. We'll discuss subroutines in exquisite detail in the chapter on composition.

The rules for flowcharting are simple, but they vary from what you may be used to seeing. Here they are:

Control diverges into two paths at a decision block and recombines into a single path at a delimiter. There is never any doubt concerning the direction of control: It flows from top to bottom. There is, therefore, no need for arrows, which would only serve to clutter a flowchart.

Don't write "THEN," "ELSE," "YES," "NO," "OK," "BAD," or other such irrelevancies on your flowchart. The THEN path of a decision is always on the right; the ELSE path, on the left. You reverse the flow control by reversing the condition in the decision block, not by representing it inconsistently in a flowchart.

Locate delimiters beneath the bottom points of the decision blocks they delimit.

Don't flowchart loop control structures in the flowchart of the loop itself. That causes confusion (a sin, according to Saint Paul) by juxtaposing loop control structures and decision structures that are part of the loop body. There only three control structures, and you can adequately express the kind of loop you have by writing its condition outside the box that represents it.

Flowchart a loop as a barred box with the control structure description written outside it. Flowchart the loop body in a separate chart. You'll see many examples of this as we proceed. Figure III-2 shows the flowcharts for the three kinds of simple loop.

Don't use flowcharting templates, ink, or plain paper to do program flowcharts. As a rule, no one else should ever see them. The best way I've found is to freehand them on quarter-inch quadrille paper. Your local office supply can sell you a 17" X 22" desk pad of this paper, which is large enough to keep the creative juices flowing. Buy the biggest eraser you can find while you're there. You'll find a pictorial example in the chapter on style.

Figure III-3 shows a flowchart for the simple IF, the most common expression of the decision structure in programming. "IF" is an abbreviation that stands for "IF-THEN-ELSE-ENDIF". Trace the logic of this chart, and be sure you understand how all the rules apply. Don't rush here. The process should take you all of seven seconds.

Programming: The Method

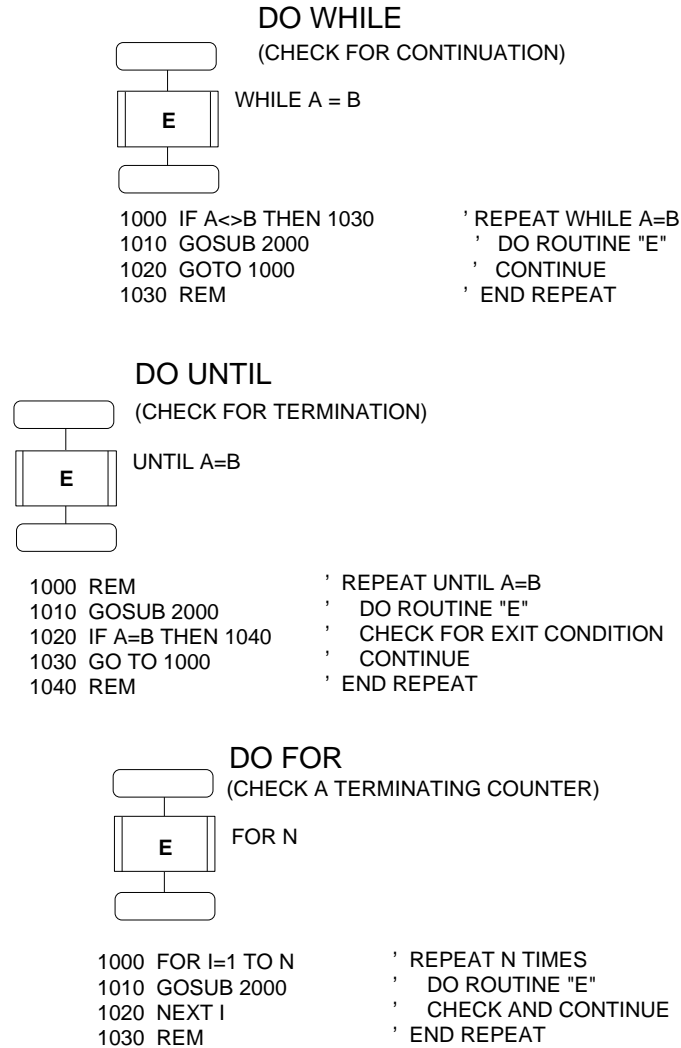


Figure III-2: The Three Simple Loops

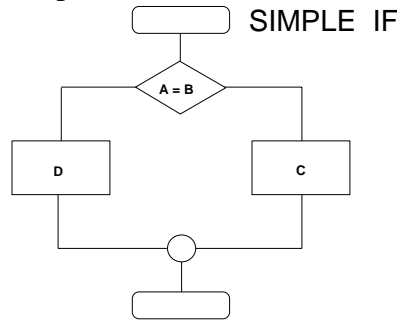
Psuedocode

Psuedocode is a description in human terms of what the computer is doing. You never separate psuedocode from the actual instructions of your program, but it doesn't just restate the instructions. Rather, it explains them in terms of the application you're trying to implement.

The rules for writing psuedocode are as simple as the flowcharting rules. You write psuedocode as a series of instructions (imperative sentences), one to a line. You show subordination, which means the same thing to BASIC as it does to Greek or Esperanto, by

indenting your psuedocode. You never indent the code itself, even if your language allows it, unless you're working with a language like COBOL that makes a certain amount of grammatically meaningless indenting ineluctable.

The instructions within the scope (between the decision block and the delimiter) of an



```
IF A = B THEN
  DO THE STUFF DENOTED BY "C"
ELSE
  DO THE STUFF DENOTED BY "D"
ENDIF
```

```
1000 IF A=B THEN 1020      ' IF A = B
1010 GOTO 1030             ' THEN
1020 GOSUB 2000            ' DO THE STUFF DENOTED BY "C"
1030 GOTO 1050             '
1040 REM                  ' ELSE
1050 GOSUB 3000            ' DO THE STUFF DENOTED BY "D"
1060 REM                  ' ENDIF
```

```
1000 IF A<>B THEN 1030   ' IF A = B THEN
1010 GOSUB 2000           ' DO THE STUFF DENOTED BY "C"
1020 GOTO 1040            '
1030 REM                  ' ELSE
1040 GOSUB 3000           ' DO THE STUFF DENOTED BY "D"
1050 REM                  ' ENDIF
```

Figure III-3: The Simple IF

IF structure are subordinate to that IF structure. The instructions executed under the control of a CASE structure are subordinate to that CASE structure. The body of a loop is subordinate to its control structure. There are no other grammatical subordinations in any programming language.

Figure III-2 shows the psuedocoding for the simple IF statement just below the flowchart. Notice that, since “IF”) is only an abbreviation for “IF–THEN–ELSE–ENDIF”, you never indent THEN, ELSE, or ENDIF under IF. You do indent anything that happens under one of these elements, though. Those are the things that are subordinate to the IF.

You can omit the process block on either side of a simple IF. A control path that goes to a delimiter without passing through a process block is called a “null path.” Programmers thus refer to “null THEN” and “null ELSE” when describing IFs of this kind. Structuralists often call null paths “stubouts,” because they represent an opportunity to put process blocks where before there were none. The ultimate stubout, then, would be one with a null THEN *and* a null ELSE, which you can certainly code if you want to.

Putting It Together

The code examples at the bottom of Figure III-2 and Figure III-3 combine coded statements from the BASIC language with psuedocoded statements explaining what they do and showing, through indentation, what their role in their respective structure is. This way of writing computer programs is the single, central notion of this book. Go over it until you thoroughly understand how it works. The process should take you anywhere from ten seconds to a whole minute.

Note that I’ve coded the IF example in two different ways. Both use the GOTO instruction to implement the structure, rather than trying to use the IF–THEN–ELSE instruction from my version of BASIC. This is because that BASIC has no delimiter (ENDIF), which makes it difficult to tell exactly which code executes in response to which condition. You can use this instruction, but you’ll end up coding the ENDIF with a GOTO anyway. It isn’t worth it to me.

The first code example uses an explicit branch (GOTO) to the THEN side of the code. This requires a GOTO to the ELSE code just after the IF. If the condition $A=B$ is false, control falls through to this GOTO, which transfers it to the ELSE.

The THEN thus consists in the GOTO followed by a REM. The REM simply gives you a place to put the word “THEN” in your psuedocode.

In the same way, the ELSE is a GOTO to the ENDIF followed by a REM. The REM simply gives you a place to put the ELSE in your psuedocode.

ENDIF is a REM that gives you a place to write your psuedocode and provides a branch location for control.

For simple IFs, you’ll probably find it convenient as well as more efficient to code according to the second example. It’s equivalent to the first, but uses fewer instructions.

You accomplish that by negating the condition in your BASIC code, but not in your psuedocode. Control thus falls through to the THEN side if the condition in your psuedocode is met, preserving the documentary quality of the psuedocode.

If you need labels for this sort of thing, you can call the first example an “explicitly coded” IF, because all parts of the statement are shown on their own lines. Of course, that makes the second example an “implicitly coded” IF.

The chief advantage of programming in this grammatical way is that it’s language-independent. This technique works for BASIC. It also works for any IBM assembler (I’ve used it in four), for any block-structured language like PASCAL or C, for any “fourth-generation” data query and manipulation language, such as SQL, and for any

macro language like the ones associated with word processors and spreadsheets. It works, in fact, for any sequence of instructions, whether you're writing a program or telling someone how to install ceramic tile in Swahili. No rule says your computer language can't be for the one between your ears.

Note at this point that I've only used one condition (“=”) in any of the example flowcharts. Of course, you can use any condition you like. If you're going to negate conditions, here are some rules for doing that:

1. Not equal is coded as “<>” – not equal.
2. Not greater than is coded as “<=” – less than or equal to.
3. Not less than is coded as “>=” – greater than or equal to.

In BASIC, you can code any variable in an IF statement. Your code will take the THEN branch if the variable is not zero and the ELSE branch if it is. Coding NOT(variable) reverses the control flow from the decision block.

Complex Structures

You don't program very long without running out of ideas you can express with the simple structures alone. You don't write Farsi poetry very long without getting beyond expressive capabilities of the simple sentence.

In your high-school English classes (You *did* pay attention in English class, didn't you?) you learned that any collection of words with at least one noun — implicit or explicit — and one verb in it is a “clause.” When you learned how to use clauses as the building blocks of sentences, you learned that there are two kinds of clauses: dependent and independent. A dependent clause is also called a “subordinate” clause when you use it in a

sentence. An independent clause is likewise called a “coordinate” clause when you use more than one of them.

A sentence that contains only one clause is a “simple” sentence. A sentence that contains at least one dependent clause is a “complex” sentence. If the sentence contains more than one independent clause, it’s a “compound” sentence. A sentence can be both complex and compound; that is, it can contain both dependent and multiple independent clauses.

In programming, every instruction is a clause. A structure that contains only a sequence of instructions is a simple structure. IFs, CASEs, and loops, therefore, are complex structures by the rules of high school English, because they subordinate instructions that perform become subordinate clauses. A simple IF, therefore, isn’t really a simple structure in the sense that a simple sentence is. The only such structure would be a single instruction that neither branches or loops.

This is unduly restrictive for programming. It’s easier to define, quite arbitrarily, a simple sequence, a conditional, or an iterative structure with only one level of subordination as a “simple structure.” This is what I’ve done.

By the same reasoning, a complex structure is one with more than one level of subordination. Such a structure is universally referred to as a “nested” structure among programmers. Simple IFs and loops aren’t thought of as nested structures, even though they subordinate other clauses in the same way. For this reason, I wanted to reserve the notion of the complex structure for application to the idea of nesting. This shouldn’t prove confusing to you as we proceed, and as you talk to more and more programmers in the course of your work.

Complex (Nested) IFs

You flowchart a nested IF by replacing any process block on the left or right side of the condition with another subordinating structure: another IF, a CASE, or a loop. Figure III-4 shows a nested IF two levels deep, along with its associated code and psuedocode. Note the two levels of indentation in the psuedocode.

Nested IFs comprise, by my observation, the third most common structure in programming. Yet they're considered so difficult to code that many programmers go to elaborate lengths to avoid them. The problem with that is that you end up trying to mask the complexity of your problem with a lot of simple, little subroutines scattered about. This is something that's harder to get right and to work with than the appropriately nested IF would be, if only you have a foolproof way of coding deeply-nested IFs. You now have such a method, meaning that you can expect to make less money if you succumb to the KISS (Keep It Simple, Stupid) method than you can expect to make with what you've just learned.

When you've finished studying this, you'll have a tool for building complex IFs to any level of nesting without losing track of where you are or what you're doing. In real life, you build structures in the flowchart, following the control line that takes you from the specified input to the specified output. As you encounter possibilities that would take you off that line, you put in decision blocks to reflect those possibilities. Later, you return to those decision blocks and chart the alternate control lines. It's as easy, intuitive, and natural as writing a letter to the editor of your local newspaper. The real trick in both cases is in knowing what you want to say.

The CASE Structure

The CASE structure of Figure III-5 is a special case of nested IF. You can build an equivalent structure by nesting IF statements with only null ELSEs.

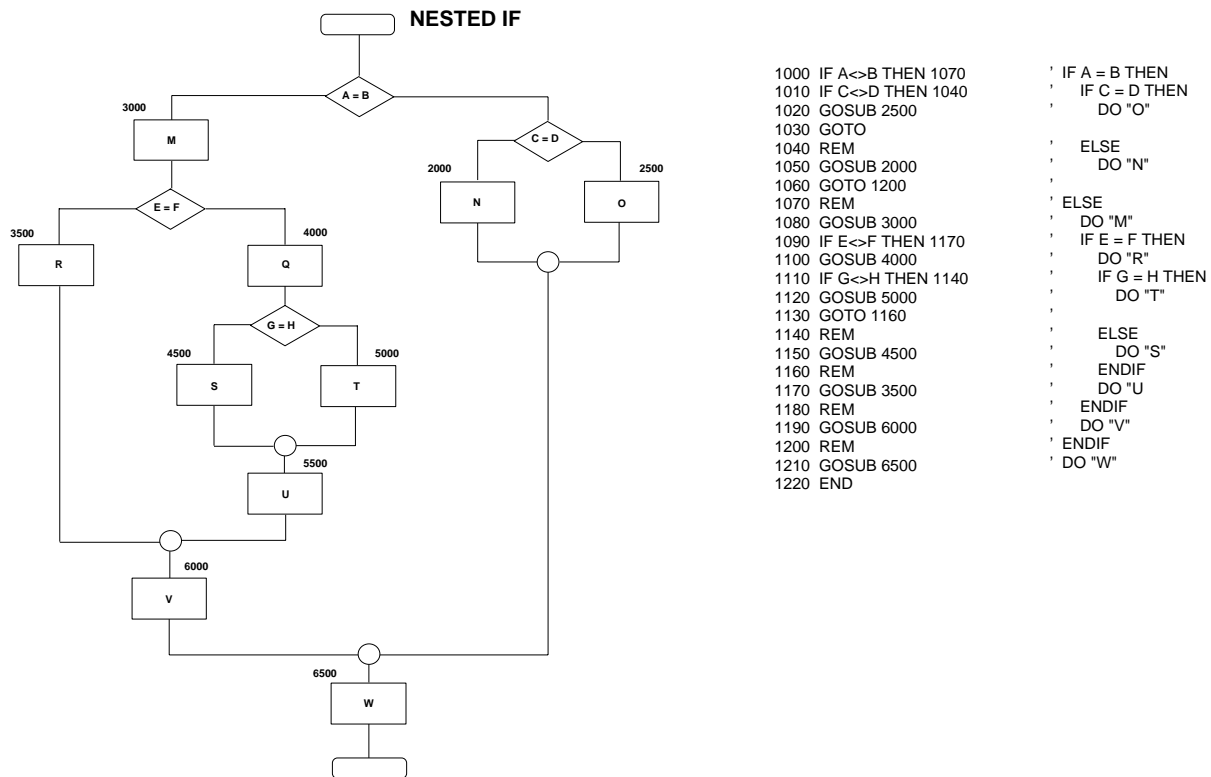


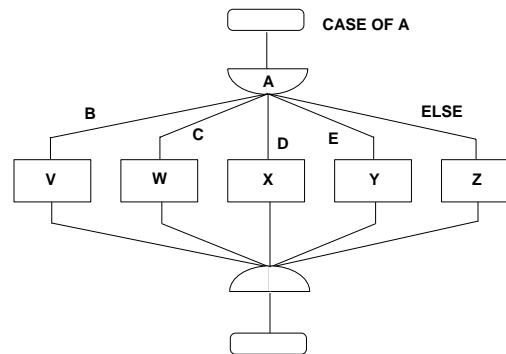
Figure III-4: A Complex (Nested) IF

You never, therefore, absolutely have to code a CASE structure, but they are convenient, and they're easier to flowchart than nested IFs. Their flowcharts also take less space.

Almost every programmer alive, moreover, understands you when you discuss the CASE structure. In my experience, it's the second most commonly coded structure in programming.

In the example of Figure III-5, A is called the "CASE variable." If A takes on any of the values written next to the control lines to the process blocks, then the corresponding process

block executes. If the value of A isn't one of the expected values, then control transfers to the ELSE process.



```

1000 REM
1010 IF A<>B THEN 1030
1020 GOSUB 2000
1020 GOTO 1130
1030 IF A<>C THEN 1060
1040 GOSUB 3000
1050 GOTO 1130
1060 IF A<>D THEN 1080
1070 GOSUB 4000
1080 GOTO 1130
1090 IF A <> E THEN
1100 GOSUB 5000
1110 GOTO 1130
1120 GOSUB 6000
1130 REM

```

```

BEGIN CASE
' IF A=B THEN
' DO THE "V" STUFF
' END
' IF A=C THEN
' DO THE "W" STUFF
' END
' IF A=D THEN
' DO THE "X" STUFF
' END
' IF A=E THEN
' DO THE "Y" STUFF
' END
' ELSE DO THE "Z" STUFF
END CASE

```

Figure III-5: The CASE Structure

BASIC has an implicit instruction, ON . . . GOTO (or GOSUB), to implement the CASE structure.

Complex (Nested) Loops

A complex loop is a loop that contains another subordinating structure entirely inside its loop body, as schematized in Figure III-6. Remember in this context that the subordinating element in a loop is its control structure.

You can't build a complex loop by nesting its control structure. The control structure, to be separate from the body structure of your loop, has to resolve to a single condition. That means that your complex loop control structure must have exactly two process blocks in it, and that these blocks can only be "CONTINUE" and "TERMINATE." If you're charting a

DO WHILE, then, the CONTINUE block is on the THEN side of your control structure flowchart. If you're charting a DO UNTIL, the CONTINUE block is on the ELSE side.

If you try to make a complex decision structure act as a loop control structure, You'll always have at least three control paths. This means that you have to repeat the TERMINATE or the CONTINUE block under one or the other decision. There is always an equivalent compound structure (see "Issues Concerning Compound Structures" below) for

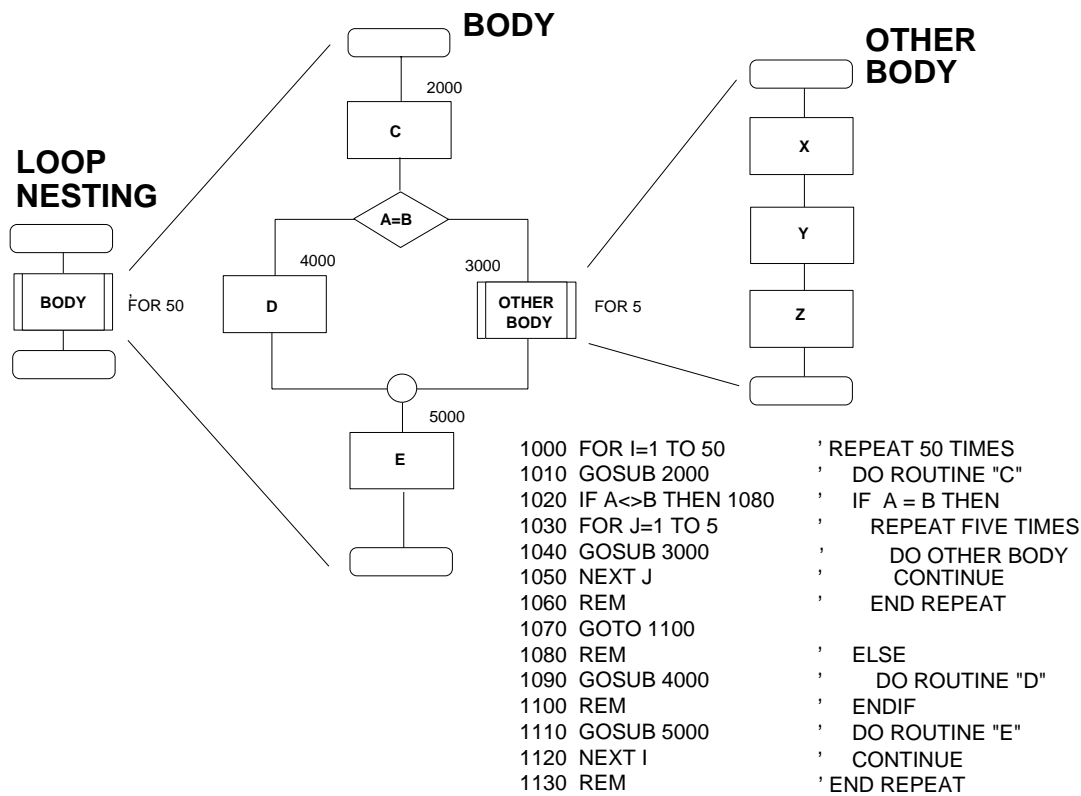


Figure III-6: A Complex (Nested) Loop Structure

nested decisions that repeat code along adjacent control paths that way.

Issues Concerning Complex Structures

When a programmer talks about "nested IFs" or "nested loops," that person is normally thinking about an IF statement inside another IF statement, or a loop inside another loop. As

far as this text is concerned, if it causes an indentation in your psuedocode past the first level, it's a complex structure by definition.

The KISS principle is not only an egregious insult to every professional programmer, it's simply stupid. I offer as an antidote to this snippet of diseased thinking Albert Einstein's dictum to the effect that a thing should be as simple as it is, and no simpler.

Don't be frightened, therefore, by complexity. Life is complex. If you can think a thing, you can write it down. If you can write it, you can program it. Anyone who tells you differently is lying to you in order to get power over you. Eschew such people.

Compound Structures

Returning to the teachings of high school English, remember that a sentence with more than one independent clause is called a "compound" sentence.

When you use them in compound sentences, independent clauses become "coordinate" clauses, because they're connected to each other by the coordinating conjunctions AND and OR.

A compound structure in programming is an IF or a loop with at least one compound condition: a sequence of conditions connected by AND and/or OR.

Compound IFs

Figure III-7 shows the simplest examples of the compound IF. To flowchart two decision blocks connected by AND, send control from the THEN side of the first decision block to the top of the second, as you would if you were nesting the blocks. Draw the ELSE line of the both blocks into the delimiter of the first block. This structure executes the THEN code if the conditions in both decision blocks are met.

Programming: The Method

To flowchart two decision blocks connected by OR, send control from the ELSE side of the first block to the top of the second. Draw the THEN lines from both blocks into the delimiter of the first block. This structure executes the THEN code if either of the two conditions is met. Remember that in programming, “or” means “one or the other or both.”

The flowcharts are mirror images of each other, as one might logically expect.

There are two ways to code a compound IF. Both are shown in the figure. The first method is an implicit coding, using the AND and OR expressions built into the BASIC IF

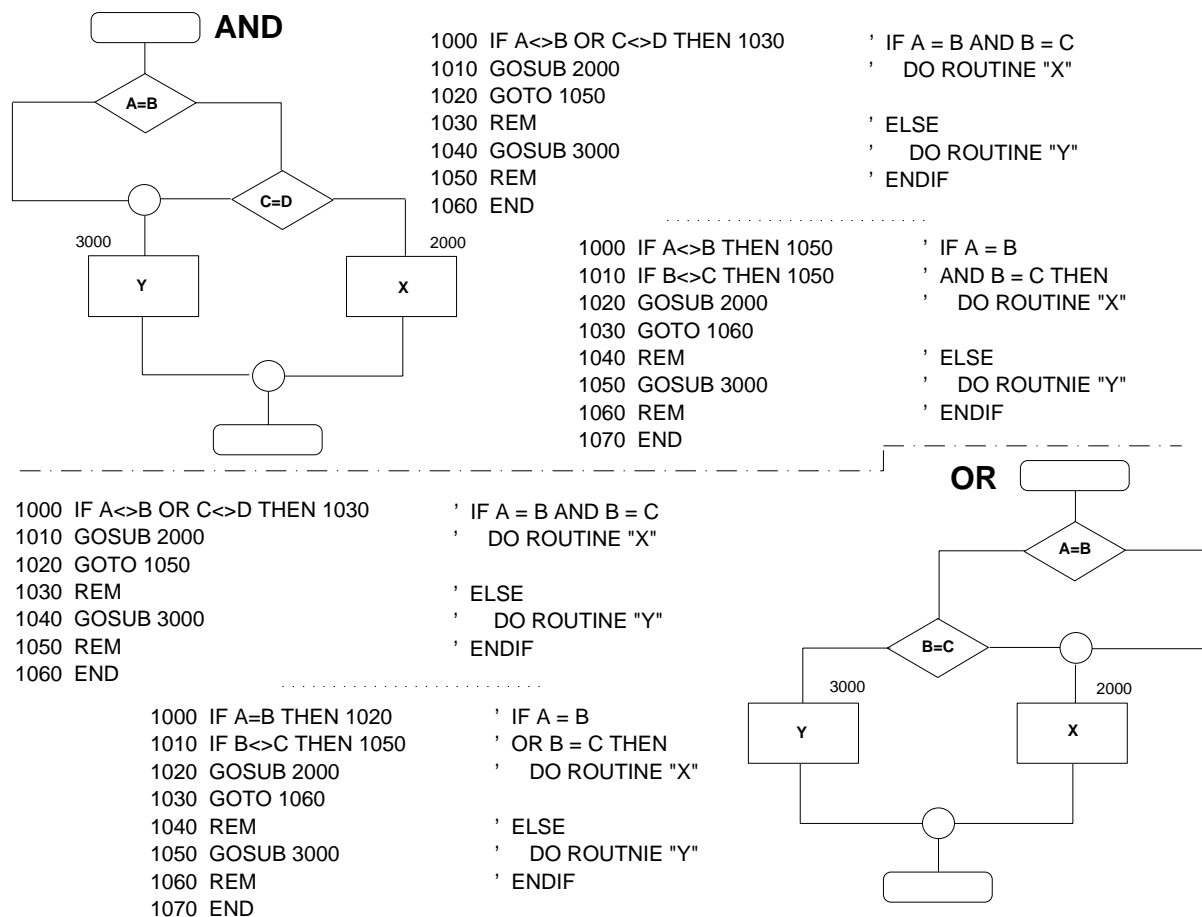


Figure III-7: Compound IFs

statement. The second is useful for “mini” versions of BASIC, or other languages like assemblers, that don’t provide AND and OR, or that don’t allow you to group coordinate

Programming: The Method

conditions with parentheses. This way of coding is also good for analyzing other peoples' programs. It's an explicit code structure that you can build strictly out of logical tests and unconditional branches in any programming language.

You can code as many coordinate decisions as you like. Figure III-8 ANDs three decisions together as an example.

Many programming languages, BASIC included, allow you to group sets of coordinate decisions with parentheses. If you want to mix ANDs and ORs, which is certainly a legitimate thing to want, you should always explicitly code parentheses in your psuedocode,

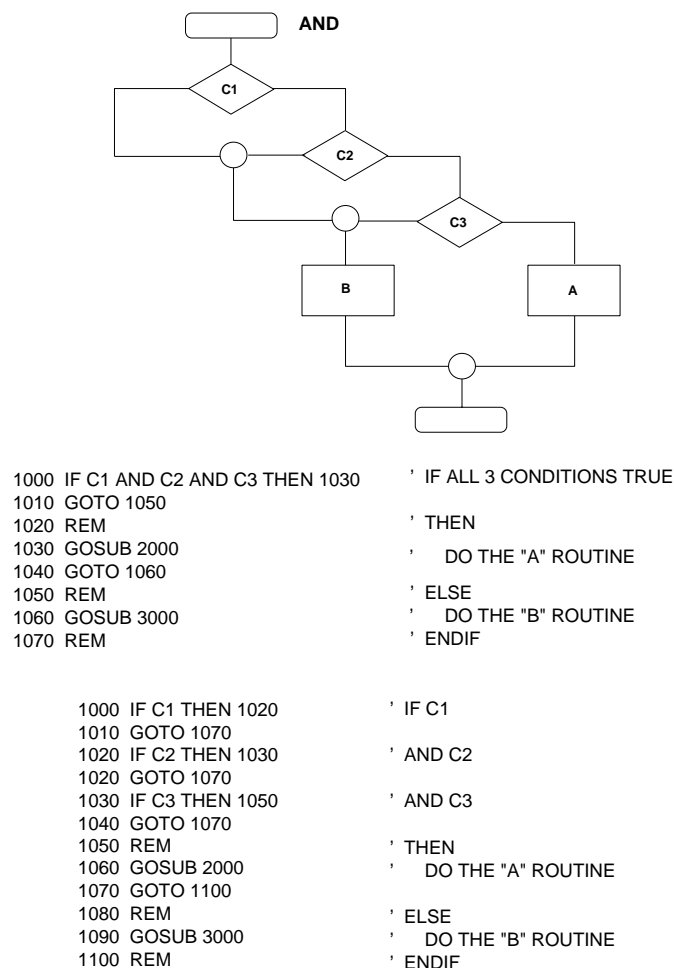


Figure III-8: Three Clauses ANDed Together

Programming: The Method

even if you know what order the comparisons will take place in. This makes it easier for you or someone else to change your program later.

BASIC performs ANDs before ORs if you don't code parentheses. If you code parentheses, BASIC performs the logical operations within the parentheses before it does anything else. Look at the two examples coded in Figure III-9. You can tell at a glance that the structures are different. With only a little practice, you can tell with only a cursory analysis just what the difference is. This ability is most helpful in maintaining existing programs.

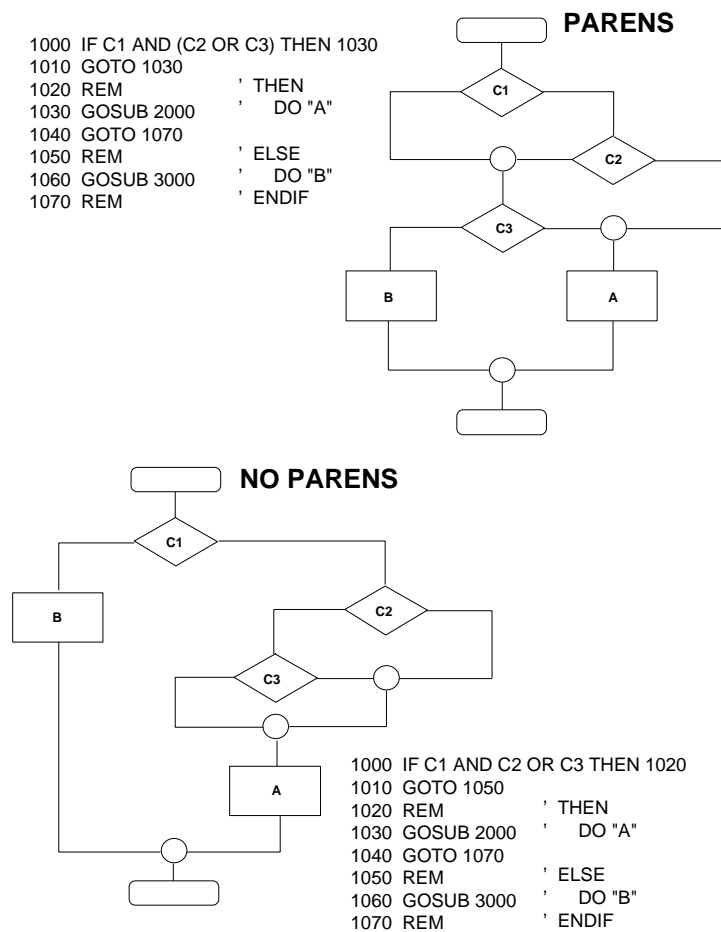


Figure III-9: Parentheses in Compound Structures

The two sets of example code are implicit and explicit, as before. Notice, however, that when I code original compound structures, I always code an explicit THEN. If you want to, you can still negate compound conditions to eliminate the extra GOTO this calls for, but as a rule, it's more trouble to negate the statement than it is to code the THEN in an original program.

If you want to negate compound statements, write the condition as a logical expression using "*" for "AND" and "+" for OR. Apply DeMorgan's laws to negate it. These laws are

$$\text{NOT}(A * B) \Leftrightarrow \text{NOT}(A) + \text{NOT}(B)$$

$$\text{NOT}(A + B) \Leftrightarrow \text{NOT}(A) * \text{NOT}(B)$$

The symbol " \Leftrightarrow " means "is equivalent to."

Negate expressions in parentheses first, then negate the outer expressions, one pair at a time.

Just remember I advised you not to write original code this way. It's another chance to make mistakes, and it only saves you two statements in your IF structure, one of which is a REM anyway. Knowing how to do it, though, is useful to the maintenance programmer.

Notice, too, that in the examples of this section, I code a variable, C1, C2, and so forth in the decision blocks. BASIC sees a variable like this as TRUE and take the THEN branch if the variable contains any value but zero. If the variable contains zero, then BASIC sees it as FALSE and take the ELSE branch.

As your compound conditions become more complicated, you'll find some other rules of value. these are

Commutative Laws

$$A*B \Leftrightarrow B*A$$

$$A+B \Leftrightarrow B+A$$

Associative Laws

$$A+(B+C) \Leftrightarrow (A+B)+C$$

$$A*(B*C) \Leftrightarrow (A*B)*C$$

Distributive Laws

$$A*(B+C) \Leftrightarrow (A*B)+(A*C)$$

$$A+(B*C) \Leftrightarrow (A+B)*(A+C)$$

You use these laws to simplify (factor) and expand compound expressions. For example, if you want to construct the flowchart of Figure III-10, you might want to flowchart it twice.

First chart the simplified expression as in the first chart. Then use the distributive laws to expand the expression, replacing the decision blocks containing the parenthesized expressions with the expansions. If you're using a language that forces you to code this structure explicitly, you have to expand it this way. But you easily can.

Coding for the example is left as an exercise.

A Compound IF with Nesting

Figure III-11 is both compound and complex. You should code this example, both explicitly and implicitly, to assure yourself that its apparent complexity offers no challenge to you.

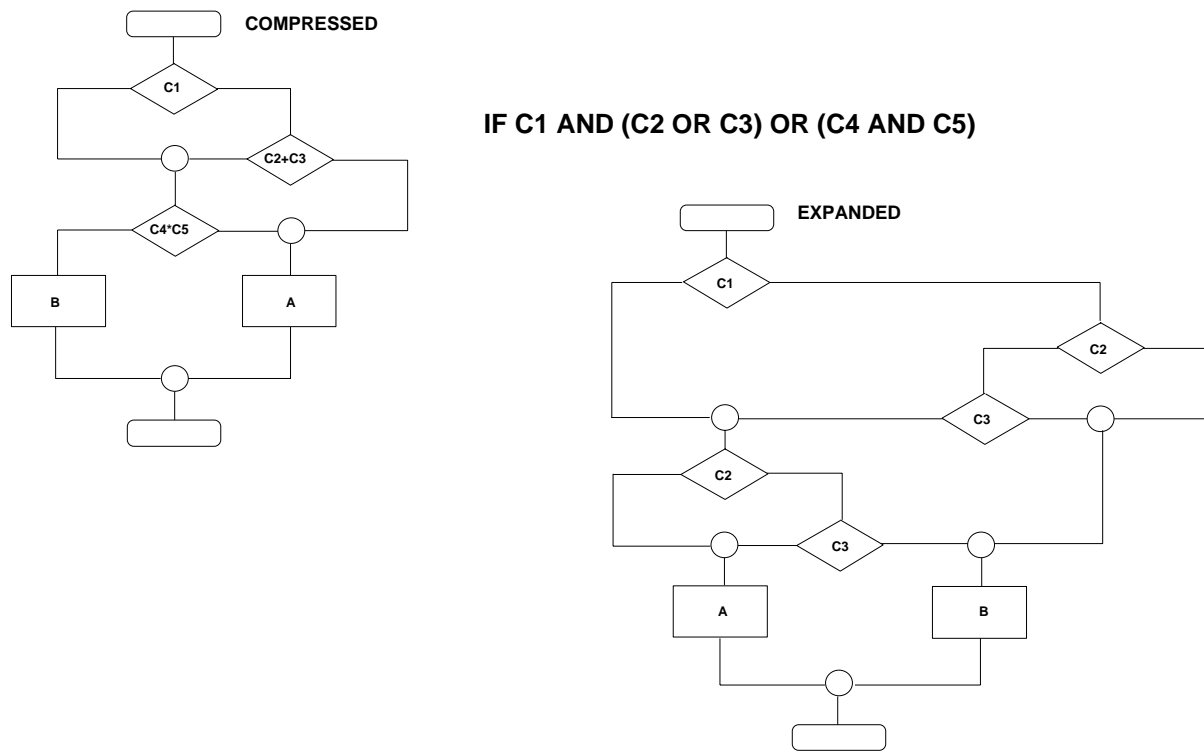


Figure III-10: A Factored and Expanded Compound IF

Compound Loops

There are two ways to compound a loop. You can compound the decision in its control structure, or you can add new control structures. You can do both in the same loop.

You flowchart compound loops in the same way you flowchart simple loops. That is, you use a barred process block to represent the body code, which you chart in a separate flowchart. You write a pseudocoded expression of the loop conditions beside the barred process block.

When you do that, you get statements that look something like this:

```
REPEAT
WHILE (C1+C2) * (MORE RECORDS IN FILE)
OR UNTIL HELLFREEZES=OVER
```

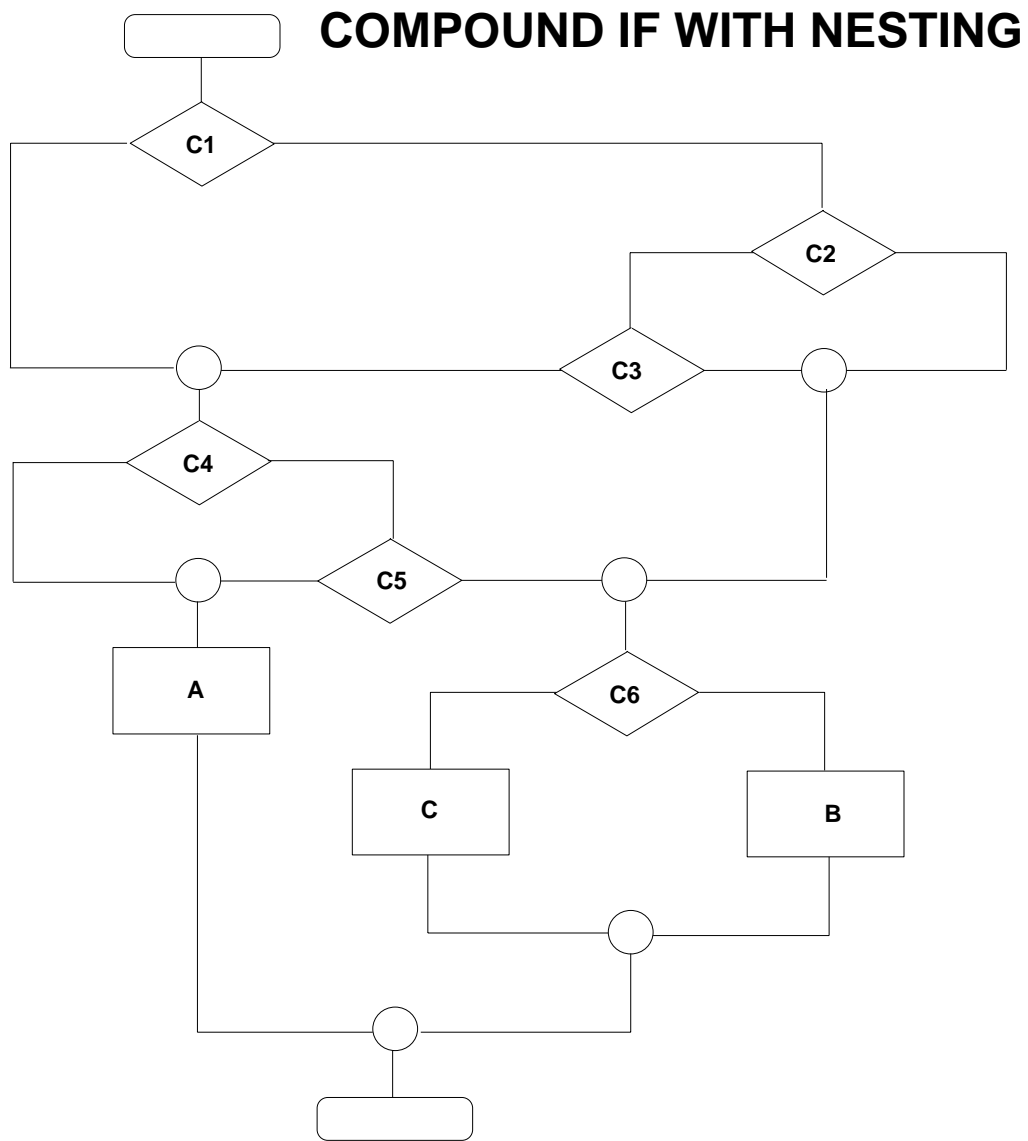


Figure III-11: A Compound/Complex IF

OR FOR A MILLION

But you can always understand them. Coding loops is easy. But then so is coding anything else, *ne c'est pas?*

If you flowchart a compound loop control structure, don't flowchart it in the same chart you map your body code in. Flowchart it separately, give it a separate name, and refer to that name in the psuedocode beside the barred box. Remember that a loop control structure always resolves itself into just two control paths: CONTINUE and TERMINATE. The chapter on program maintenance contains an loop charted like this in the input loop example.

Issues Concerning Compound Structures

What happens when you stick a process block on one of the vertical control lines in a compound IF, as in Figure III-12?

The structure you get is certainly thinkable, and therefore grammatical. One way to express the condition of the flowchart is as an ablative absolute in the following:

```
IF CHICKENS ARE PRESENT
AND, THE FOX KILLED, THE COAST IS CLEAR
THEN
    COUNT YOUR BLESSINGS
ELSE
    COUNT YOUR EGGS
ENDIF
```

What do you do, though, when the process block occurs on the ELSE side of the first decision? I don't know a good way to express the result in English.

I call these structures "conundra," because I don't know what to do with them.

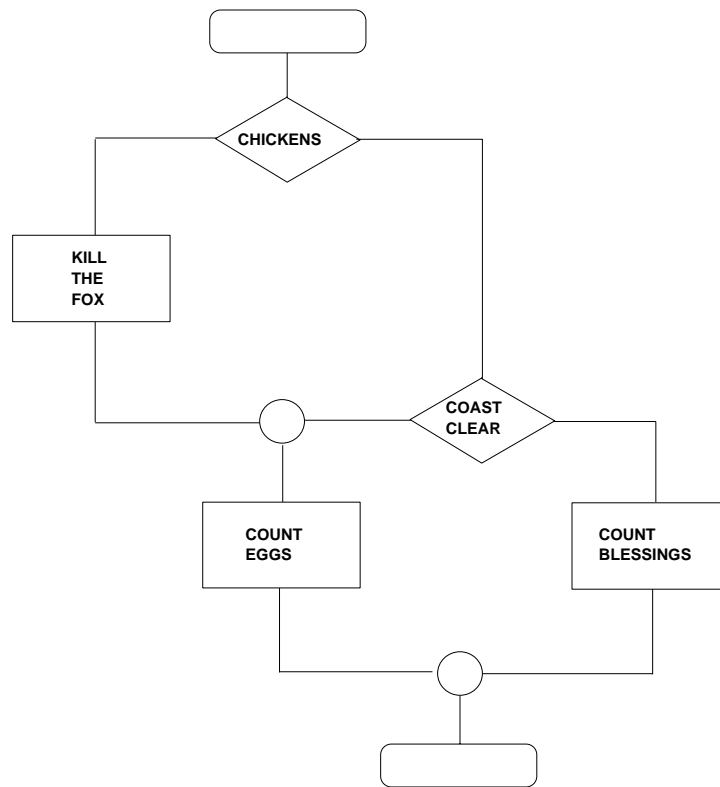
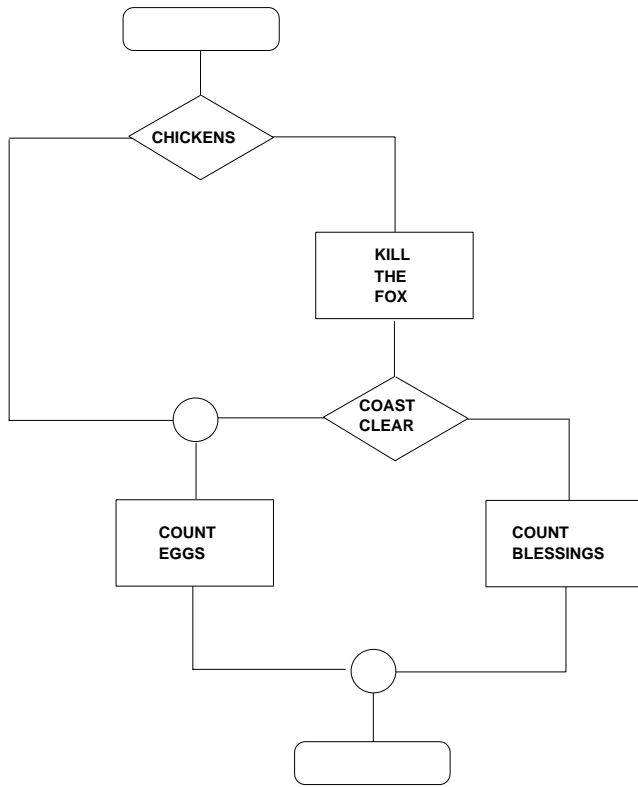


Figure III-12: Conundra

You most often get these structures when you analyze another programmer's code. This is because these structures are really two structures compacted together into something that looks like a single structure. The equivalent flowchart appears in Figure III-13.

I'm not completely satisfied with this, believing as I do that there must be a way to code this idea that doesn't repeat the decision block. Still, this is the best I've been able to do. You may wish to pioneer in this area. All it represents is the limit to which I can take you. That hardly implies limits on how far you can go.

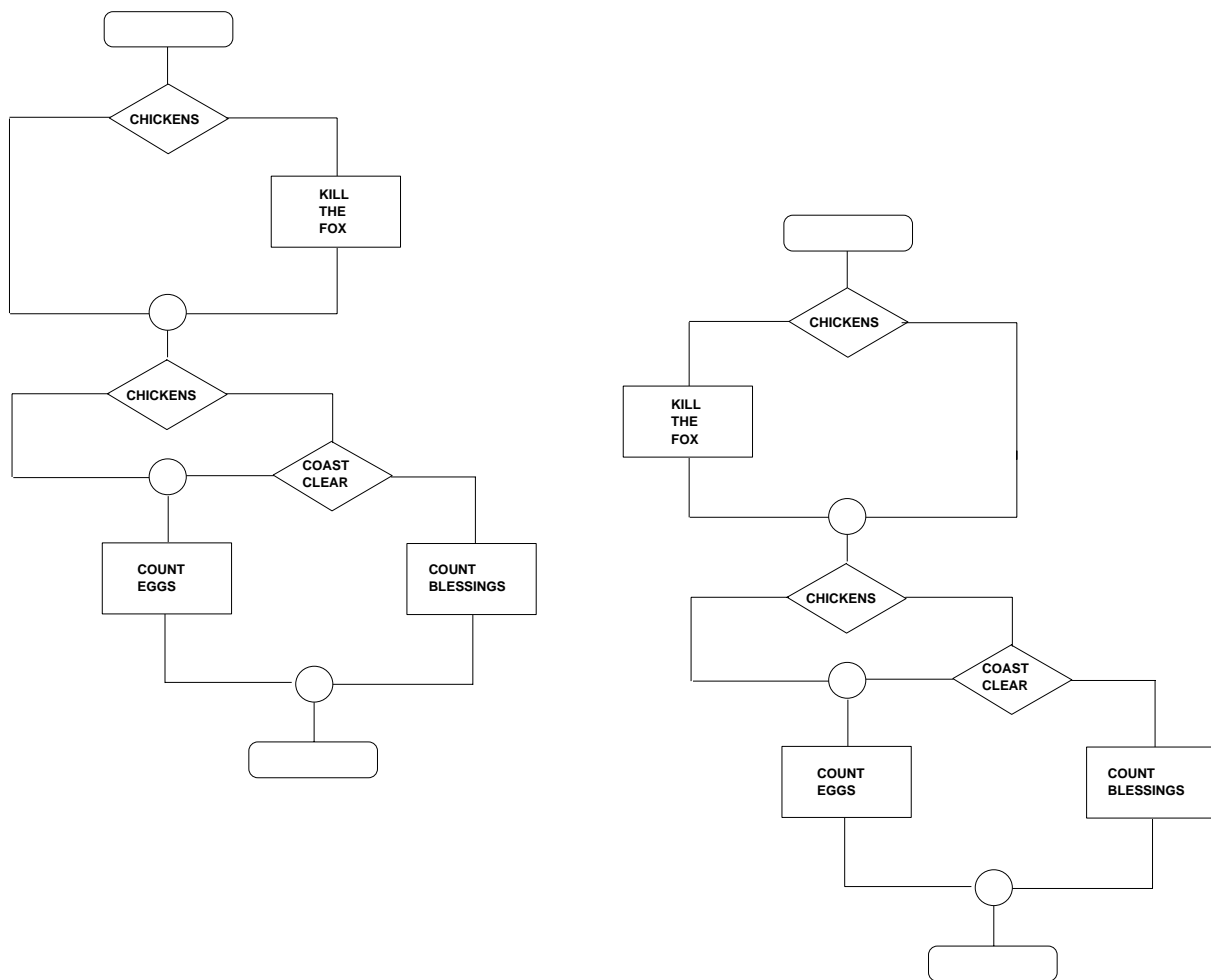


Figure III-13: An Alternate Expression for Conundra

There's a flowchart that maps an existing structure of this kind in the spaghetti code example of the chapter on maintenance (Figure IV-3). For reasons I discuss in that chapter, you rarely if ever see a similar structure more complicated than this one that actually works in production.

Finally, consider the flowchart of Figure III-14. It repeats the same code along two adjacent control paths.

Any time you get a structure like this — and you will as you use flowcharts to analyze other peoples' programs — what you have is the equivalent compound structure in the same figure.

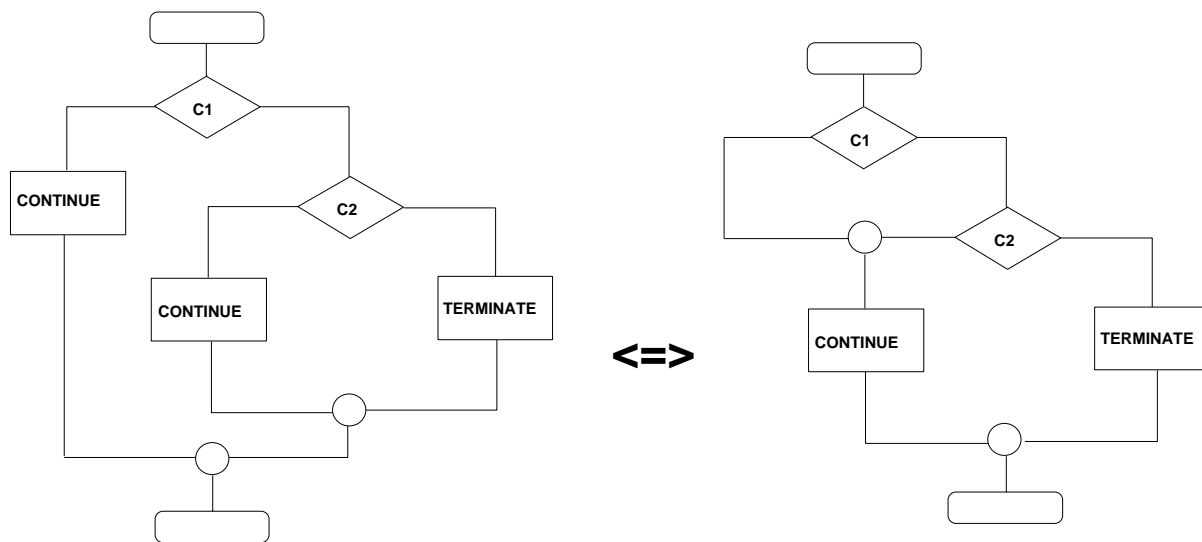


Figure III-14: Repeating Code Along Adjacent Control Paths

What if the paths aren't adjacent, as in Figure III-15? In that case, you can get an equivalent structure that does meet the criterion for a compound condition by negating one of

the conditions in the chart. This is why you can't make a complex loop by nesting its control structure. If you try, you always end up with a structure like this one.

There is one loop structure I haven't dealt with. This is the recursive loop, which is a

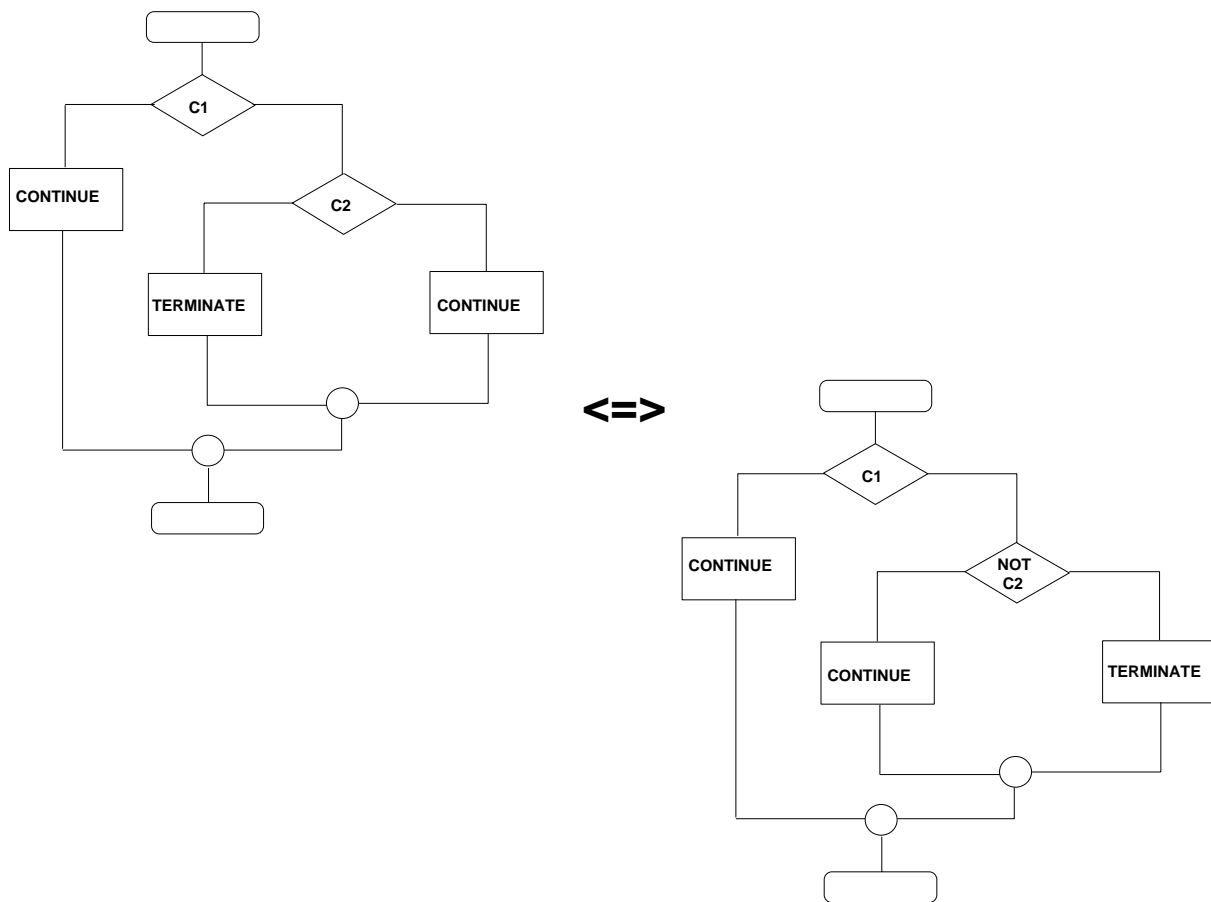


Figure III-15: Forcing Adjacent Control Paths

loop that transfers control from its body to some point in its initialization structure. These loops do exist, but you rarely encounter them. You can always find an equivalent structure, and many languages popular in professional programming circles don't let you code recursive loops. If you ever do have to maintain one of these things, you'll almost always find it worth your time to recode it into one of the normal structures. Recursive loops are

Programming: The Method

another area in which you might wish to transcend the limitations of this book. Who am I to naysay?

Chapter IV: Grammar for Maintenance

If you've read this far, you know that only your imagination limits your ability to write computer programs. If you can think it, you can program it. Now you're ready to do it for money.

What you'll likely discover is that almost all work being done in programming today is maintenance programming. You get a cubicle, a terminal, and a system that you alone are responsible for keeping up. Users go to you or your manager with requests to have the system changed. Perhaps they want a report the system used to put out but doesn't any longer. Maybe they've changed the way they track payments and need a dozen databases altered. Sometimes you have to bug hunt, or alter two hundred programs to recognize leap year every four years. It really happens.

All this seems routine and boring until you discover how hard it is to do it well. When you went to computer school, wherever that was, you probably learned a lot of computer language syntax, and not too much else. Perhaps you got a soupcon of "structured" programming, an undifferentiated, prescriptive amalgam of grammar, composition, and style. But the people whose code you're maintaining probably didn't go to the same school. Their programs aren't "structured" at all, at least as you understand the term. It's not long before a lot of beginning programmers find themselves floundering.

Most programmers who survive this initiation view it as just that. They promise themselves that they'll someday be managers, or system administrators, or systems analysts — anything but programmers. As soon as they can, they get out of programming altogether.

This is both bad for the art and a mistake for the programmers who do it. Study the life of any happy, successful person, and you'll discover someone who did the jobs no one else wanted with industry, class, and panache. The maintenance programmer who can't wait for morning to bring a new day's challenges can always count on having a job. When new opportunities emerge, these programmers stand at the head of the line.

The structuralists — they represent a coherent, political and social movement — have never explained to me what an “unstructured” program should look like, or how such a program can possibly work if it exists. They're willing to tell me how to program only to the extent that I'm willing to reject as invalid every computer program written before around 1980, and about ninety percent of all programs written since. Clearly, if there is an answer to the maintenance programmer's lament, it isn't in structured programming.

If a program runs, consistently producing a specified output from a specified input, doesn't it stand to reason that most of its structures should be easily describable in terms of the grammar you use for creating programs? They are. In fact, they're simpler than programs you're able to write using the methods of the last chapter. Maintenance programming is as easy and satisfying as any job in data processing if you respect the ability and intelligence of the people who preceded you, bending your efforts to finding out what makes their programs work.

Analyzing Structures

Flowcharts provide the most useful tool for maintenance programming. They let you examine structures without rewriting them.

You analyze structures by flowcharting equivalent, normal structures. Almost all of the structures you'll actually deal with are simple IFs and loops. Now and again, you'll

encounter a nested structure. Compound structures are common, but rarely extend to more than a couple of conditions.

There are some structures we haven't discussed, but you can easily find normal equivalents to them. You already know everything you'll ever need to know about flowcharting.

To analyze a program, then, the first thing you have to do is isolate the structural components from the functional components of your program. If you have long, simple sequences, begin by compressing them into single blocks in a flowchart. Wherever your computer checks a condition (as in the IF instruction of BASIC), see whether you have a simple IF or loop that can be viewed as wholly subordinate to another structure. If it can, compress that simple structure into a single, functional flowchart box. You can expand it later.

Identifying IFs

This compression done, you identify IFs and CASEs by examining forward branches. A forward branch is one that sends control *down* the coded page. If a conditional branch (BASIC IF statement) sends control to the statement following an unconditional branch (BASIC GOTO), then you're looking at an IF-THEN-ELSE-ENDIF structure. If a structure contains a number of forward branches to a common location, then you've often got a nested IF or CASE structure, the common location being the implied ENDIF for the every structure subordinate to the nesting.

Almost all real-world IFs work by negating their conditions to fall through to the THEN code, branching explicitly to the ENDIF or the ELSE. This is also true of compound IFs. As you flowchart and psuedocode these structures, then, you'll find DeMorgan's laws helpful, even though you might not use this method of building IFs in your original programs.

If you encounter a series of decisions that branch to other decisions without intervening structures, you probably have a compound IF. Let's look at some examples from real life, which I've translated into BASIC from actual, working assembler and COBOL programs.

Simple IFs

You deal with simple IF structures all the time. This is a typical example.

```
1000 IF A<>B THEN 1030 ' IS THIS A REGULAR SALESPERSON?
1010 GOSUB 2000        ' NO, GIVE THEM A 2 PERCENT BONUS
1020 GOTO 1040
1030 GOSUB 3000        ' YES, WRITE SORRY LETTER
1040 IF C$=LEFT$(SOCSEC$,3) THEN END
```

Consider that you can have a dozen statements or more between the "NO" comment and the "YES" comment, and that these comments are never indented, you can see why some programmers might consider this kind of structure confusing. Instruction number 1040 delimits this IF, but there's no indication of that in the comment fields.

Still this is a perfectly normal structure. You might code it as follows:

```
1000 IF A<>B THEN 1030 ' IF THIS IS THE SALESMAN OF THE MONTH
1010 GOSUB 2000        ' COMPUTE A BONUS PER CURRENT TABLE
1020 GOTO 1040
1025 REM              ' ELSE
1030 GOSUB 3000        ' WRITE A CONSOLING LETTER
1035 REM              ' ENDIF
1040 IF C$=LEFT$(SOCSEC$,3) THEN END
```

The flowchart for both code sequences is exactly the same. The only coding difference between the two blocks is the two REM statements I added to the second block to hang structural tags on. You usually don't have time to recode or recomment existing structures, though. The average data processing shop still has over a year's worth of backlogged projects. That's what makes the flowchart so valuable: It tells you what the program is doing

without making any changes to the code at all. Changes you don't make don't have to be tested, and that saves you time.

Complex IFs

For an IF to be complex, it has to subordinate an entire IF or an entire loop (except, maybe, for its initialization structure) to THEN or ELSE. As a rule, nested IFs result from exception checking. You should look for them, then, around I/O operations, because those are the operations that most commonly generate exceptions.

An INPUT # instruction, say, might fail for any number of reasons: a file not found, for example, or an attempt to read past the end of the file. If the instruction completes successfully, you might have invalid data in the record, depending on how carefully your data entry routines edit the information you type into them. Here's a real-life example:

```
1000 IF EOF(1) THEN 1160           ' RECORDS ALL PROCESSED?
1010 LINE INPUT #3,X$             ' NO - GET THE NEXT RECORD
1020 IF ERR THEN 1130             ' DID THE READ FAIL?
1030 Y=VAL(LEFT$(X$,9))           ' NO - CONVERT PAY AMOUNT
1040 IF Y>=1000000 THEN 1110      ' MORE THAN A MILLION?
1050 Z=VAL(MID$(X$,6))            ' NO - CONVERT FED TAX
1060 IF Z=0 OR Z=Y THEN 1080      ' IS THIS UNREASONABLE?
1070 GOSUB 2000                   ' NOW GO PROCESS PAYROLL
1075 GOTO 1150                     ' . . . KEEP ON TRUCKING
1080 PRINT #4,"BAD TAX AMT - SKIPPING"
1100 GOTO 1150
1110 PRINT #4,"PAY MORE THAN A MILLION $ - SKIPPING"
1120 GOTO 1150
1130 PRINT #4,"READ FAILED - PROBLEM WITH INPUT FILE - DYING"
1140 END
1150 GOTO 1000      ' NOW GO BACK AND GET ANOTHER RECORD
1160 GOTO 1140     ' ALL DONE!
```

There are actually several potentially confusing things going on in this example. The first BASIC IF statement isn't really an IF at all, but a control structure statement for the loop that ends with the GOTO in statement 1150. The second decision controls the printing of a read failure message, but it also compounds the loop control structure: The composite loop

condition thus becomes REPEAT UNTIL NO MORE RECORDS AND WHILE READS ARE VALID. Statement 1150 serves as the delimiter for the two loop control decisions as well as for the two nested IF decisions in the loop body proper, plus the compounding decision that shares the loop body function of printing the error message.

Some of the confusion here comes from the nature of an I/O statement. The INPUT # instruction doesn't simply tell the computer to bring something into memory from the disk. It tells the computer to bring something into memory from the disk and set a condition. The condition is TRUE if the I/O fails for some reason, FALSE otherwise.

That means the INPUT # instruction has to precede the loop control function, even though it's also a function of the loop body. Likewise, the printing of the error message is also a loop body function, but it's triggered by the error condition set in the INPUT # instruction. You can't fully separate the body code from the control structure in this kind of operation, but you can flowchart the two structures separately.

The curious way this example ends, leaving the loop only to bounce back into it to terminate the program, is an example of "spaghetti code." We'll discuss the spaghetti structure — and it is a valid structure — later.

The flowchart of Figure IV-1 shows the structure of the example. You should try at this point to reproduce the chart without looking at the figure. That should prepare you for what we plan to do with this piece of code in the section on normalizing structures. Take a few minutes to be sure you've got it right. If you have a problem, stop and look at the figure, then start over. It's really very important to understand this example completely before you go on.

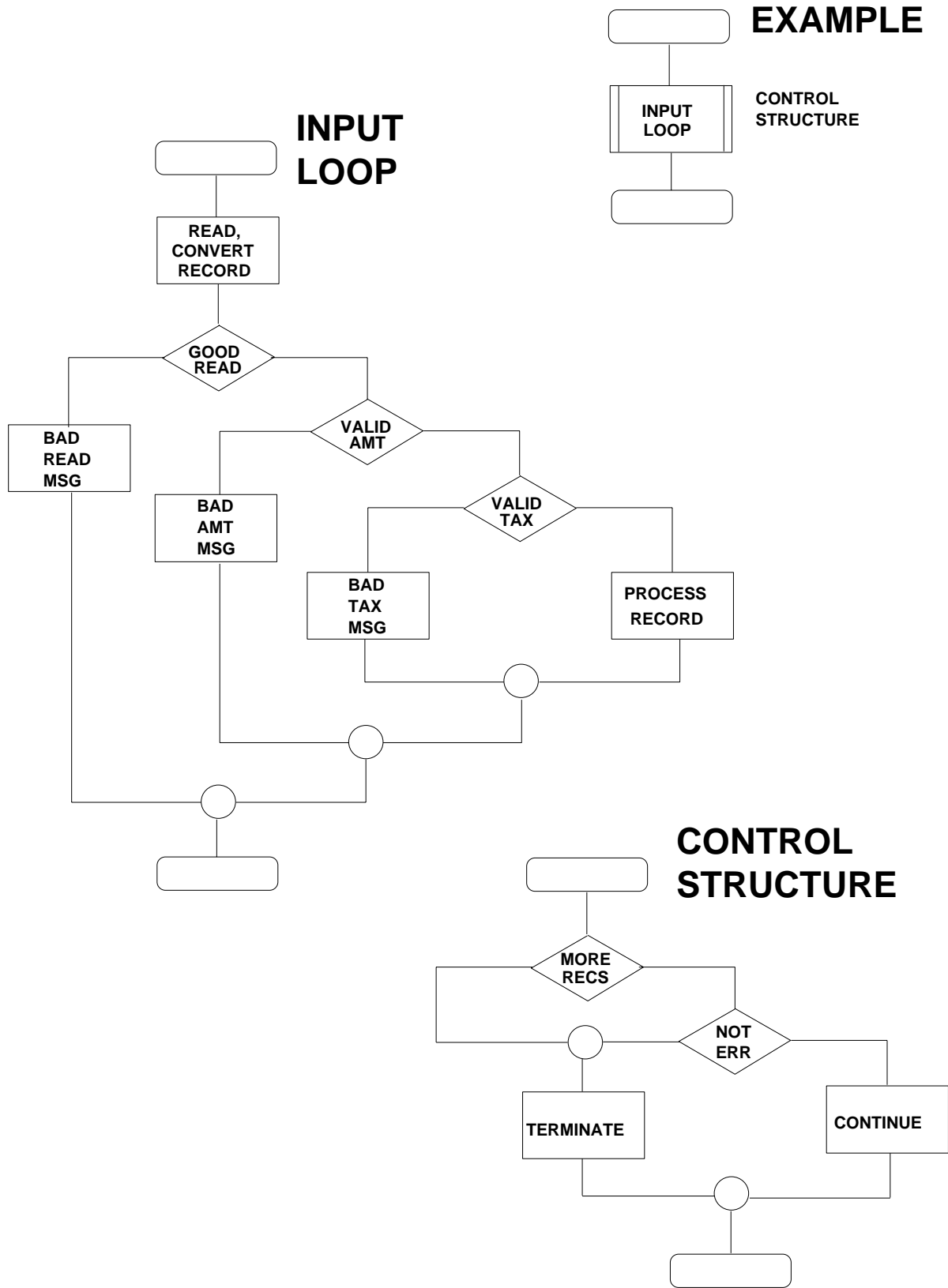


Figure IV-1: Flowchart for the Input Loop Example

Identifying Loops

You identify loops, as a rule, by examining backward branches. When you see a branch to a lower-numbered statement, such as statement 1150 in the example, you're probably headed back to the first statement in a loop control structure. Analyzing the loop from that point becomes a matter of isolating that structure from the body code. If you thoroughly analyzed and understood the loop structure in the example, you've seen just how this works.

Most loop structures in data processing are loops that get a record, do something to it, write it out, then get the next record. You call these "input loops."

Normalizing Structures

The term "normalization" refers to finding normal equivalents for things that work but aren't in a form that everyone who works with them agrees on and understands. Normalizing structures, therefore, is the expression of programming structures as something everyone agrees on.

Input Loops with Branchbacks

You've already done a certain amount of this when you flowchart a module like the one in the example using the standard flowcharting methods we began with. Alas, I lied to you when I said I drew the example strictly from real life. I did, but I rewrote some of its branches to avoid flooding you with too much information at once. Here's what the code originally looked like:

Programming: The Method

```
1000 IF EOF(1) THEN 1140           ' RECORDS ALL PROCESSED?
1010 LINE INPUT #3,X$             ' NO - GET THE NEXT RECORD
1020 IF ERR THEN 1130             ' DID THE READ FAIL? QUIT.
1030 Y=VAL(LEFT$(X$,9))           ' NO - CONVERT PAY AMOUNT
1040 IF Y>=1000000 THEN 1110      ' MORE THAN A MILLION?
1050 Z=VAL(MID$(X$,6))            ' NO - CONVERT FED TAX
1060 IF Z=0 OR Z=Y THEN 1080      ' IS THIS UNREASONABLE?
1070 GOSUB 2000                    ' NOW GO PROCESS PAYROLL
1075 GOTO 1000                     ' . . . KEEP ON TRUCKING
1080 PRINT #4,"BAD TAX AMOUNT - SKIPPING"
1100 GOTO 1000
1110 PRINT #4,"PAY MORE THAN A MILLION $ - SKIPPING"
1120 GOTO 1000
1130 PRINT #4,"READ FAILED - PROBLEM WITH INPUT FILE - DYING
1140 END                           ' ALL DONE!
```

Satisfy yourself that the second example is equivalent to the first: That is, for any given input, both code blocks produce the same output.

In general, two structures are equivalent if and only if they produce the same flowchart. Trace the flow of information to assure yourself that the single flowchart of Figure IV-1 describes both structures. This means that even though neither structure is “normal” as we’ve defined the term, both are equivalent to one that is. If you want to, then, you can code a normal equivalent to these examples. In real life, although you’ll want to make the flowchart, you won’t have time for recoding. But then, recoding won’t be necessary, either.

The difference between the second example and the first is that it skips records by branching back to the beginning of the loop control structure. This makes the body code seem to share functions with the loop control structure.

Note that the one normal backward branch in line 1150 of the first example is missing from the second. Its functions have been relegated to body code. If it were there, it would never execute. Note also that the spaghetti code is missing: There’s no reason to leave the loop except to terminate the program.

This is a very common way of coding. Programmers do it because they've had no reason to distinguish between loop control structures and loop bodies. Thus, they have no problem branching to either the top or the bottom of an input loop when they want to skip a record.

Nevertheless, the backward branches this generates don't perforce become part of the loop control structure. They *should* be forward branches that implement IFs in the loop body. The function of the backward branches should be collected into a single backward branch, corresponding to line 1150 in the first example, that follows the last ENDIF of the body code.

Once you understand that, normalizing input loops becomes a snap. All you have to do is isolate the "real" control structure from the spurious control functions the backward branches generate. You do this by condensing the body code, backward branches and all, into a single process block on your flowchart. Now you can flowchart the loop as a barred process block with the control structure written in words beside it, implicitly replacing the backward branches with a single branch from the bottom to the top of the loop.

Now consider the code you separated into the process block. That's the loop body. Flowchart it by mentally reversing the backward branches to a single, forward point, which is a combined ENDIF for all of the decisions of that block. If you mess up and get some of the control structure mixed in with your body code, don't worry about it. You won't be able to flowchart the structure you get doing that, which will just make you rethink the two loop structures until you get them properly separated.

In the real world, just about everything that programmers find confusing stems from interspersing loop control functions throughout the body of a loop. This happens to be a very common example, but here are some others:

Multiple Loop Exits

You get multiple loop exits by transferring control from the body of a loop to more than one location outside the loop, as in the following:

```
1000 FOR I = 1 TO 50          ' FIND THE FIRST BLANK OR ZERO
1010 K$=MID$(X$,I,1)
1020 IF K$ = " " THEN 1020
1030 IF K$ = "0" THEN 1040
1040 NEXT I
1045 GOTO 1050
1020 GOSUB 2000              ' BLANK? DO FIRST PROCESS
1030 GOTO 1050
1040 GOSUB 2500              ' ZERO? DO THE SECOND PROCESS
1050 END
```

The flowchart of Figure IV-2 maps an equivalent structure.

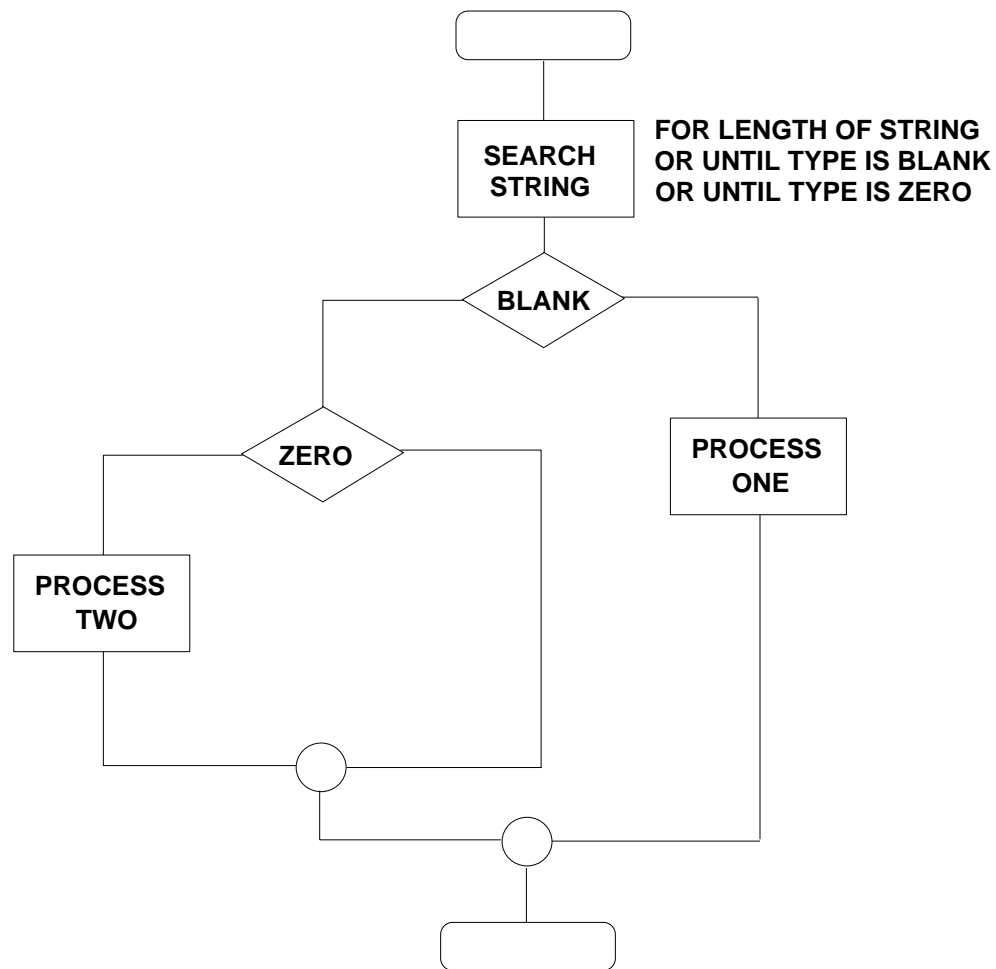


Figure IV-2: The Multiple Loop Exit Equivalent

Spaghetti Code

You get spaghetti code by writing a backward branch that doesn't implement a loop structure. This is confusing, but it's neither ungrammatical nor "unstructured." You can't write a program that works at all if it simply lets control wander aimlessly about. If programmers, on the other hand, deliberately build spaghetti code, as one must assume they do, and if the computer reliably executes that code, then one has to be able to describe it in terms of normal structures. If it were not so, the grammar itself, not the program, would be inadequate to its task.

The tricky part of writing executable spaghetti code is to execute the instructions between the unconditional branch and its target. Here's a real-world example of such a structure translated from an assembler program that has run without a hitch since 1974:

```
100 I = 1;FLAG$="OFF"  
110 IF TABENTRY$(I) = IPVALUE$ THEN 140  
120 IF TABENTRY$(I) = "THAT'S ALL, FOLKS" THEN 180  
125 I=I+1  
130 GOTO 100  
140 FLAG$ = "ON"  
150 GOTO 210  
160 J=1  
170 IF TABENTRY2$(J) = IPVALUE$ THEN 140  
180 IF TABENTRY2$(J) = "AIN'T NO MORE!" THEN 210  
190 J=J+1  
200 GOTO 170  
210 END
```

This code block uses an input character string to search two tables, turning a flag on if the input string matches any of the values in either table. When you follow control instruction by instruction to flowchart something like this, you typically get a flowchart that looks like the one in Figure IV-3. Do this one yourself until you fully understand how you get this chart.

Notice that when I drew this flowchart as a nested IF, I had to repeat the same code on the same side of two nested decision blocks. As we discussed in the previous chapter, any

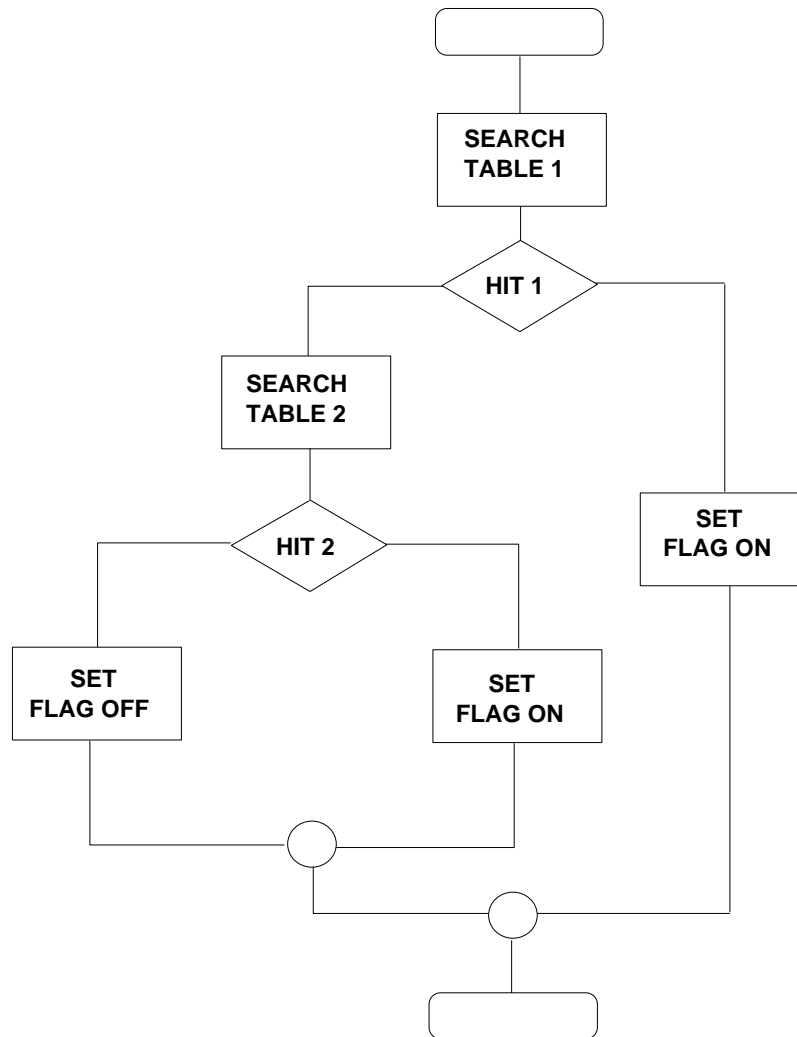


Figure IV-3: The Spaghetti Code Equivalent

time you get this kind of flowchart, what you have isn't a complex structure at all, but a compound structure or a conundrum. When I get a conundrum in a flowchart like this, I usually leave it in the nested form. I'm not satisfied with that, either, by the way.

The Compleat Grammarian

The second listing, YASVPTP, of the example program in Appendix A is grammatically the same as the first example. It's there to give you some practice untangling structures that typify the kind of things you'll encounter as a maintenance programmer. By

“grammatically the same,” I mean that both program versions run to normal completion, producing exactly the same output from the same input file.

The hardest temptation to resist when you’re analyzing someone else’s code is the temptation to invest that code with greater complexity than it actually incorporates. One of the charges that structuralists — who long ago made their own escape from the programmer’s life — bring against programmers is that we can’t solve difficult problems with computers because we can’t code difficult structures.

You now know that this simply isn’t so. You have the ability to code the most byzantine, convoluted solutions that mere human life can call for without ever getting lost or even terribly confused.

Your predecessors, however, didn’t know that. They stayed pretty much within the scope of the simple IF and the constraints of much-copied input loops. If they coded complex structures at all, they usually didn’t nest more than a level or two. They built the odd spaghetti structure, but as a logical challenge, spaghetti code doesn’t amount to much. They avoided compounding wherever they could, because they had no really good way of drawing pictures of compound structures, as you do by now.

Above all, they copied. If you know a structure works and you want to incorporate it in a new program, you should by all means do so. You’re getting paid for the amount of your code the computer executes, not the amount that you write or that other people read. Your deathless, guiding principle should be, “Never make anything you can buy. Never buy anything you can get for nothing.” You can’t copyright an idea, after all, which means you really can’t steal a piece of program logic.

Programming: The Method

The difference between you and all those others is that you can now use other peoples' logic intelligently. When you write a new program, there's no rule that says you can't use someone else's idea. There's also no rule that says you can't recode that idea a way that makes your work easy for the people who follow after you to understand. If you do that, I predict you'll die a legend.

Chapter V: Composition

Composition describes the arrangement of sentences into paragraphs and of paragraphs into written works. Programmers refer to paragraphs as “modules,” “routines,” or “subroutines.” “Programs” are groupings of modules that produce a specified input from a specified output. When you study composition, then, you’re learning how programs are organized.

In your first college English course, your instructor probably concentrated on teaching you enough of the fundamentals to get you through school. The idea was to help you write a good laboratory report, or to complete an essay examination successfully. Because writing programs is so much simpler than writing in spoken languages, if you can remember what you learned in your first few sessions of English 101, then you know everything you’ll ever need to know to compose programs.

If you never went to college, don’t worry about it. The next few paragraphs contain everything your instructor would have told you about programming.

The Parthenon Form of the Essay

The parthenon offers a popular mnemonic for English composition teachers. A picture of it appears in Figure V-1.

To write an essay in this form, you first express the thing you’re going to write about in a single sentence, called a “thesis sentence.” The thesis sentence contains, as a rule, only one independent clause and any number of dependent clauses.

The independent clause is what you think. The dependent clauses abbreviate the reasons you think so. You build an essay first by stating the thesis, then by writing a series of

supporting paragraphs, each of which begins with a sentence you build out of the dependent clauses of your thesis sentence.

This done, you write a conclusion that restates your thesis sentence, but comes to a

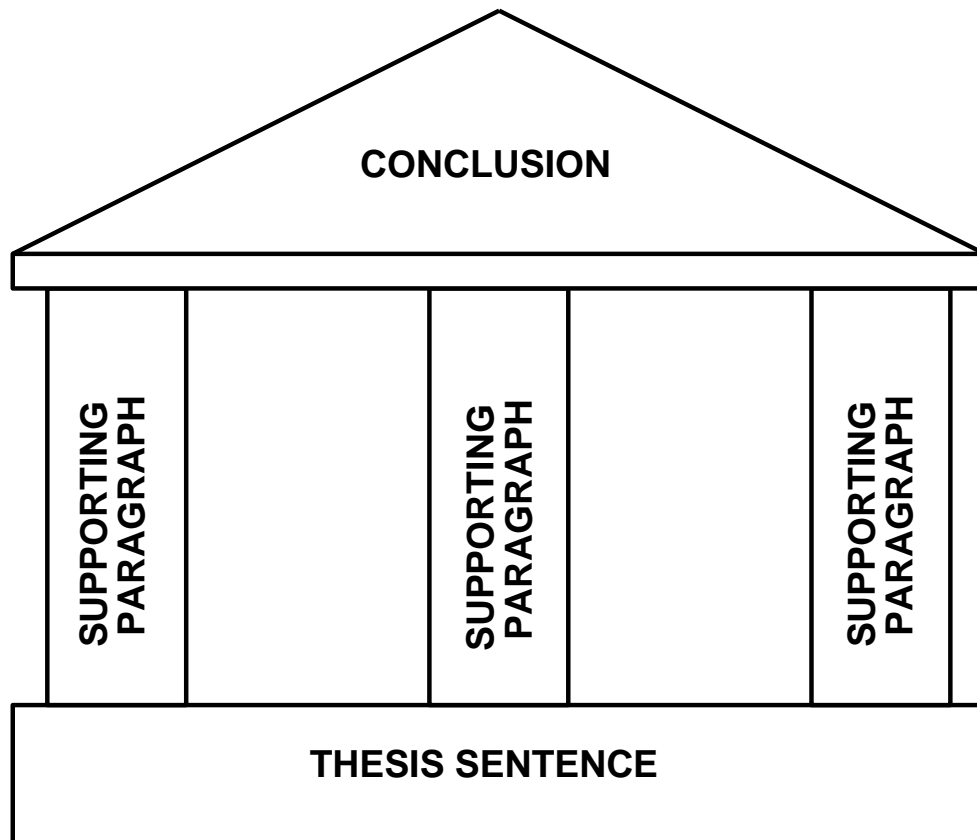


Figure V-1: The Parthenon Form of the Essay

point: an epiphany, a call to action, or some such.

Public speakers use this form a lot. Their adage is, “tell ’em what you’re going to say, say it, then tell ’em what you said.” The reason speakers find this form so useful is that it infallibly provides fastest way to structure a speech, especially one that you have to deliver within a set time and for which you haven’t prepared.

Programming: The Method

Writers use it less, because they consider it naive. Still, when you have to get something out in a hurry, you can usually do a better job with the parthenon form than with any other kind of organization. That's why it's so useful for essay exams and lab reports. Here's an example:

Thesis: Soma should remain illegal because it makes you stupid, because it causes itching of and unsightly hair growth on the palms, and because it causes incarceration of the entire body as a physical side effect.

Supporting Paragraph: Soma makes you stupid. In the experience of thousands of people who have tried it, soma can be counted on to make your big toe infinitely fascinating: You can sit and stare at for hours. Surely, you have a better use for your time than that!

Supporting Paragraph: Soma causes itching of and unsightly hair growth on the palms. In areas like California and Florida, where palms are more common than they are, say, in Wisconsin, soma is thus a greater problem. But do we not all suffer in the end?

Supporting Paragraph: Soma causes incarceration of the entire body for as long as ten years in this state. This adverse physical side effect alone should serve as a stern warning to users of this dangerous substance: If the effect develops, it's definitely time to quit.

Conclusion: Given these evils, then, we must not only keep this drug illegal, we must take stringent measures to eliminate the use of soma wherever it springs up. Write to your legislator to have it declared America's first National Public Nuisance!

Clearly, it doesn't matter what you write, as long as you organize it properly.

This way of subordinating many little ideas to one big one is what makes the human brain powerful. It lets you to keep track of any number of details under that single umbrella

notion without having to concentrate on the details or worrying about getting lost in them. It keeps you focused.

The Parthenon Form of the Program

You don't write programs to move a computer's heart or to charm another programmer with the elegance of your expression. Your conclusions are likely to consist in closing a couple of files and executing some statement asking the computer to flush your opus from its memory.

You write programs primarily for a machine that will deal with them in a few, peremptory nanoseconds. The computer does not take time to appreciate your compositional style. Neither do the humans who have to maintain your work once you've committed it to production. If it fails, they will deal with it, one line at a time, at four in the morning. They will think of you, but not to admire the way you write. Given all that, does it make sense to use anything but the simplest, most straightforward composition you can that works? If you agree that it does not, then you'll use the parthenon form exclusively for composing programs.

You'll have a lot of company. This form is so compelling that programmers from the time of ENIAC have relied almost exclusively on it. Indeed, the greatest contribution of the "structured programming" movement came in this area, as the structuralists labored to comprehend the fundamentals of freshman composition.

Building Modules

You build program modules out of structures the same way you build paragraphs out of sentences. The idea is to isolate the functional elements of your program, forming single, functional units out of them.

Programming: The Method

Just as a paragraph expresses a single, however complex, thought, a module does only one job. What constitutes “one job” is a decision that you make as a programmer, hence a matter of style. Still, if you give any two programmers the same piece of code and ask them to shape it into modules, they’ll usually agree within an instruction or two. Use judgement.

Programmers generally delimit and document modules with starred boxes. In BASIC, a starred box commenting a module looks like this:

```
1000 REM *****
1010 REM * THIS STARRED BOX CONTAINS AN EXPLANATION OF THE SINGLE *
1020 REM * PROGRAM FUNCTION OF THE MODULE THAT FOLLOWS. *
1030 REM *****
1040 IF FLAG% OR A$=B$ THEN XXXX 'IF FLAG OFF AND A<>B THEN
1050 C$="" ' WIPE OUT C
1060 GOTO
1070 REM 'ELSE
1070 C$=LEFT$(B$,9) ' GET THE PAYROLL AMOUNT
1080 C=VAL(C$) ' CONVERT IT
1090 REM 'ENDIF
1100 REM *****
1110 REM * THIS STARRED BOX CONTAINS AN EXPLANATION OF THE SINGLE *
1120 REM * PROGRAM FUNCTION OF THE NEXT MODULE IN THIS SEQUENCE. *
1130 REM *****
1140 CLOSE #1
1150 CLOSE #2
1160 END
```

There are two ways of arranging modules in a program. You can write them as a series of simple sequences — a simple sequence of modules — as in the example, or you can write them as “subroutines.”

Subroutines are code blocks that execute outside of the main control flow of the program. A subroutine “call” stores the address of the instruction following the one that issues the call, then transfers control to the first instruction of the subroutine. The subroutine, which may itself call other subroutines, completes, then executes a “return”, which retrieves the address saved in the call and transfers control to it.

In BASIC, the subroutine call is the GOSUB . . . RETURN instruction. The general understanding among programmers is that a subroutine is a single module as we describe it here. This is because you don't know when you write a program whether you'll want to call a subroutine from more than one location later. If you code two or three functions in a single routine, then you quickly get tangled up trying to separate them.

You have to be careful here. It's easy to make a program read to another human being like a book that's written in footnotes. Some languages encourage this kind of coding syntactically. On the other hand, if you try to keep too many modules on the same level, you can make them overlap in a frightfully confusing way, as you'll see in the real-life example of this chapter.

Building Programs

You build a parthenon form program by organizing its modules into something called a "functional hierarchy." This is nothing but an organization chart in which the higher-level modules call modules running at lower levels. Called modules are therefore subordinate to calling modules, just as supporting paragraphs are subordinate to thesis sentences in freshman essays. Figure V-2 is the functional hierarchy for the example program of Appendix A.

Notice that the term "called module" refers only to a module that executes under the control of another module. "Module" does not mean "subroutine," any more than "paragraph" means "footnote."

Programmers refer to the module that calls, directly or indirectly, every other module of the program the "main" or "mainline" routine. Typically, the mainline code incorporates an "initialization" module, the function of which is to open and close files, set variables to

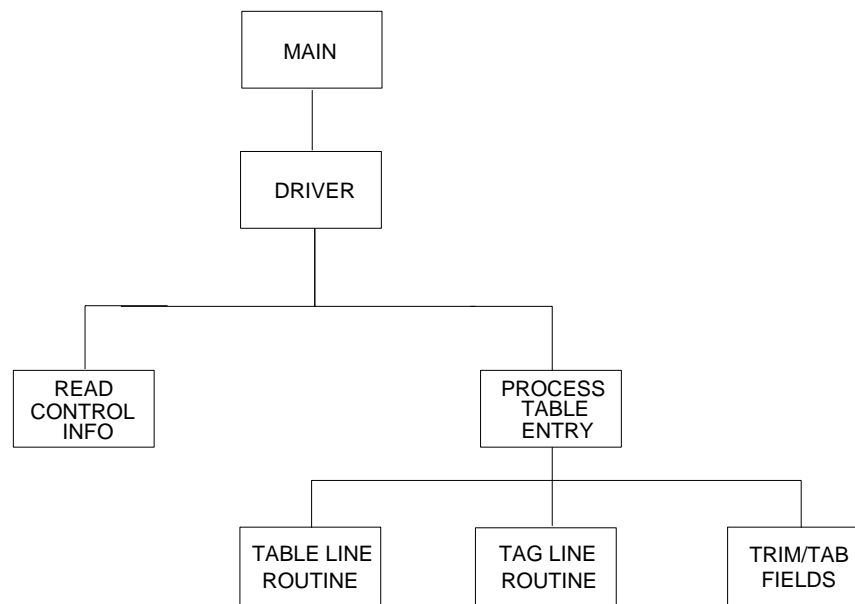


Figure V-2: A Functional Hierarchy

initial values, and perform other functions that your program does only once each time you run it. The main module of a program thus consists mostly in subroutine calls.

When you design a program, you should begin with a written specification of its input and output. That gives you a clear statement of what you want to accomplish, at least when you set out. Be aware that users are notorious for changing their minds, and that by the time you get the program written, your user will almost certainly have done so. That's because business changes. If you've specified your input and output conscientiously, though, and if you've diligently applied the other programming techniques you mastered in this text, you'll be able to change your program as fast as your user's world turns over.

The second thing you do is compose the program, listing the functions it has to perform to get the specified output from the specified input, and arranging them into a functional hierarchy.

You then design the most complicated modules by flowcharting the structures they contain. Don't worry about the simple stuff: You can code that off the top of your head when you get to it.

When you finally do get around to coding it, the hard part is all done. Let the functional hierarchy determine the order in which you write your modules, psuedocode as you follow your flowcharts, describe each module in a starred box as you proceed, and the code will literally stream from your flying fingers.

You'll end up with a fully-functional, fully-documented program that once debugged stays debugged, and that you or anyone else can easily change to suit your most mutable customer's heaviest demand. You now lack but one skill to be a professional programmer.

Analyzing Composition

The one thing remaining is to be able to analyze the composition of existing programs in the same way you analyze their structures. Remember that composition, like grammar, is descriptive, meaning that you can express any program that works as a functional hierarchy.

This also means, incidentally, that you can't change the composition of a working program except to make it unintelligible, hence destroying it as a program. You can, within limits, move modules around, relegate some structures to subroutines, and so forth. But the functional hierarchy for the program won't change as you do that.

Consider the following real-world example translated from its native assembler:

```
1000 REM *****
1010 REM *           I N I T I A L I Z A T I O N           *
1020 REM *****
1030 OPEN "PGM.CTL" FOR INPUT AS #1      ' OPEN CONTROL FILE
1035 OPEN "LPT1:" FOR OUTPUT AS #4      ' OPEN PRINT DEVICE
1040 LINE INPUT #1,X$                  ' GET SWITCH VALUE FM CTL
1045 UPSISW%=VAL(LEFT$(X$,1))          ' SET VALUE OF UPSI SWITCH
```

Programming: The Method

```
1050 IF UPSISW%=192 THEN          ' IS SWITCH INVALID? QUIT
1060 CNTRL$="0"                   ' NO - SET SEQUENTIAL MODE
1070 PAGESIZE%=58                 ' SET PRINTER PAGE LENGTH
1080 IF UPSISW%<>64 THEN 1100     ' IS SELECTIVE PRINT REQ'D
1090 LINE INPUT #1,X$            ' GET THE NEXT CONTROL CARD
1100 STATUS$="0"                 ' SET STATUS TO ZERO
1110 GOSUB 5000                  ' OPEN INPUT AS RND OR SEQ
1120 IF CNTRL$<>"0" THEN 1400     ' IF SEQUENTIAL BRANCH
1130 IF UPSISW%=128 THEN 1500    ' IF RANDOM, BRANCH
1140 STATUS$="3"                 ' SET SEQUENTIAL MODE
1150 LINE INPUT #2,Y$            '   FOR BEGINNING OF FILE
1160 GOTO 1400                   ' BRANCH TO RANDOM
1170 PRINT #4,"CONTROL FILE SPECIFIES SEQUENTIAL INPUT"
1180 PRINT #4,"AND SELECTIVE PRINT - INVALID COMBINATION"
1190 CLOSE 1,2,4
1200 END
1210 REM *****
1220 REM *           M A I N L I N E   L O G I C           *
1230 REM *****
1400 KEY1$=MID$(X$,2,5)          ' SET UP KEY
1410 IF CNTRL$="1" THEN 1500     ' BRANCH IF RANDOM ACCESS
1420 STATUS$="2"                 ' READ FIRST RECORD IN RANGE
1430 GOSUB 5000
1440 GOTO 1520
1500 STATUS$="1"                 ' READ NEXT RECORD
1510 IF CNTRL$"1" THEN 1530
1520 STATUS$="0"                 ' RANDOM ACCESS
1530 GOSUB 5000                 ' READ RECORD
1550 IF STATUS$="0" THEN 1600    ' BRANCH IF FOUND
1560 PRINT #4,"NO RECORD FOUND"
1570 LINE INPUT #1,X$            ' GET ANOTHER CONTROL CARD
1580 GOTO 1400
1600 GOSUB 7000                  ' SET UP PRINT REPORTS
1610 REM *****
1620 REM *           S E C T I O N   O N E           *
1630 REM *****
1650 REM     HERE FOLLOWS A LONG SEQUENCE OF MODULES, TITLED
1660 REM     "SECTION ONE" THROUGH "SECTION FIVE." THIS ENTIRE
1670 REM     CODE BLOCK CAN BE VIEWED AS A SINGLE, SIMPLE SEQUENCE
1680 REM     BROKEN UP INTO FUNCTIONAL MODULES BY STARRED BOXES.
1690 REM     "SECTION SIX" IS INTERESTING FOR THE WAY IT ENDS.
1700 REM *****
1710 REM *           S E C T I O N   S I X           *
1720 REM *****
1730 REM     THIS SECTION BUMPS AND GRINDS, MOVING AND PRINTING DATA.
1740 REM     IT THEN TERMINATES AS FOLLOWS:
1750 IF CNTRL$="0" THEN 1500     ' GET ANOTHER RECORD IF SEQUENTIAL
1760 IF CNTRL$="1" THEN 1800     ' GET ANOTHER CONTROL RECORD
1770 N%=N%+2
1780 IF MID$(X$,N,2)="XX" THEN 1500 ' IF NOT COMPLETE GET NEXT RECD
1800 IF EOF(1) THEN 1200        ' ALL DONE? BRANCH
1805 LINE INPUT #1,X$            ' GET ANOTHER CONTROL RECORD
1810 GOTO 1400                   ' CONTINUE
```

If the program encounters a fatal error somewhere in one of the omitted modules, it branches to the END instruction in statement number 1200, as it does when it runs out of information. This means that the modules at 1400 and 1500 are input loop control structures that return control to the END statement upon completing.

The key structure in this program is an overlapping loop, which you build by the way you compose the program, rather than the way you build the program's individual structures.

The programmer who wrote this writes structures that are extremely easy to read and analyze. Yet the program as a whole is confusing. That happened, if my sources inform me aright, because the programmer was responding to a shop standard that says, "Thou shalt not code as subroutines the chief work of thy program." This is a very common standard among software vendors and data processing shops, and there are several good reasons for standards somewhat like this one.

This program, however, first mistakes the standard as it's really intended, then applies the mistaken rule perversely. To find out what this program really does, examine the flowchart of Figure V-3 as you analyze the code of the first module.

The misdirection springs from the starred boxes, which only purport to describe what's happening. The "mainline" module isn't the main module at all, but rather two, overlapping subroutine loops separated logically by the convoluted control structure of the first and last modules.

The "initialization" routine does indeed initialize the program, serving as the initialization substructure for one or the other loop. Then it issues a subroutine call, cleverly disguised as an unconditional branch, to one of them depending on the initial conditions. Note that only one of the loops executes in each execution of the program, and that either

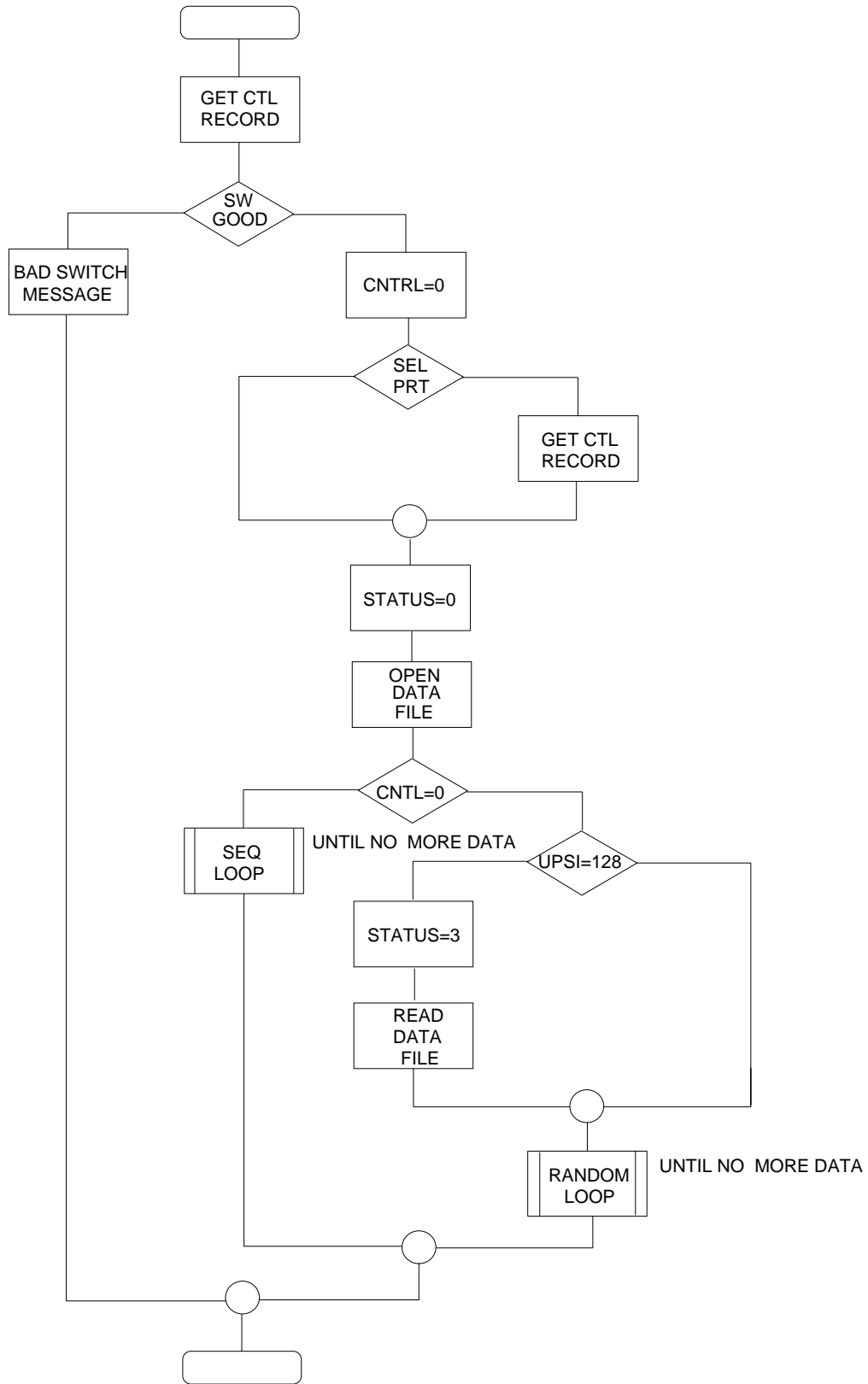


Figure V-3: Flowchart of the Overlapping Loop Example

loop returns control to the “initialization” routine’s END statement on completion, disguising with equal cleverness the subroutine return. That makes “initialization” the mainline module for this program.

“SECTION ONE” through “SECTION SIX” constitute the overlapped loop body structures. If you take these down a level in the functional heirarchy, you can skeletize the fused loop control structures, isolating them for further analysis. The flowchart, therefore, relegates these sections to barred process boxes to be dealt with later. The resultant functional hierarchy appears in Figure V-4. This describes the actual composition of the program.

Notice that if more than one calling routine calls a single subroutine, the subroutine’s block shows up in the functional hierarchy more than once. This is how you know what level you need to put the called routines on: They have to be at least one level lower than any routine calling them.

Almost always, if you have trouble understanding a program that merges its structural elements in this way, you have a problem in composition. Analyze it by isolating its body structures into barred boxes to separate the control structures. In this example, you have to consider what conditions are always met when control transfers to the loop from the initialization structure. These belong to the second structure, and can be eliminated from your consideration of the first.

In the end, all these decisions boil down to the loop control structures written beside the two barred boxes. Trace the flowchart a few times, comparing it to the code, until you see exactly how this works.

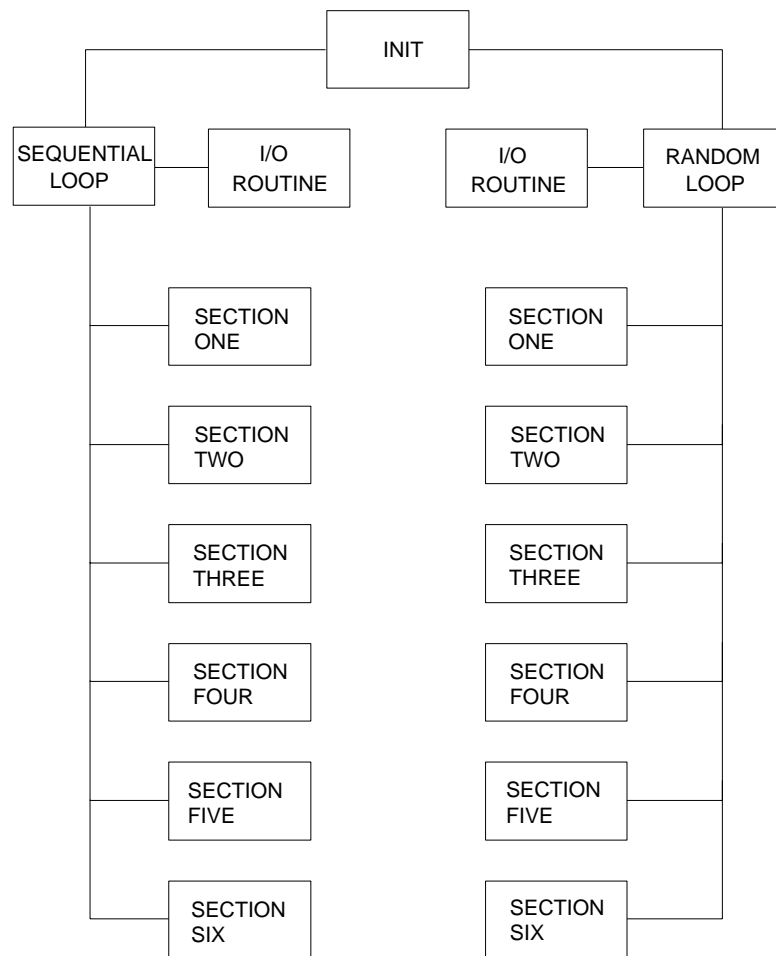


Figure V-4: Functional Hierarchy for the Overlapping Loop

Don't worry about encountering a program that's more complicated than this one. It's so difficult to compose a program that actually works this way that few programmers even try. Remember that if it doesn't work, it isn't a program. Not even a poorly composed one.

By the same token, if you find yourself coding something that becomes ever more problematic, drawing you deeper and deeper into a tangle of convoluted logic, you did something wrong when you composed the program in the first place. Stop, recompose the program, then throw away what you've already coded and start over. If that doesn't work, go back to your input and output specifications: You've misconceived the problem you were

Programming: The Method

trying to solve. Programming is easy. It's supposed to be. It can be difficult only when you try to do it wrong.

Chapter VI: Style

Most of the methods of the chapter on coding original programs are really prescriptions of style. It is, I hope by now, obvious that you can write working programs that don't conform to the style, and that you can use the style itself to describe these programs. If you don't believe it, try to code a program that produces a specified output from a specified input, but that you can't describe using the normal structures.

The idea is to free you to solve problems using the computer as easily and as capably as you'd use a ruler and string to lay out your garden, or pencil and paper to write a novel. A tool you use well is one you don't notice as the thing you make emerges. A good tool is easy to use well.

Grammatical Style

The more you do a thing, the more sophisticated your style becomes. An advanced style, therefore, is the product of hundreds of repetitions, rather than the odd revelation or stroke of genius. This is why a professional's work seems always to look different from an amateur's, even if the two pieces accomplish the same thing. The more programs you write, the better they work and the easier they are to maintain. It's a matter of style.

Flowcharting Style

The easiest way to make flowcharts according to the stylistic prescriptions of Chapter III is to freehand them on quarter-inch quadrille paper. You'd be surprised how easy it is to think in quarter-inch squares.

Figure VI-1 shows more or less what such a flowchart looks like, except that I never do real flowcharts on the computer. A 1" X 1/2" process block seems just big enough to write inside without driving me off the edge of my paper.

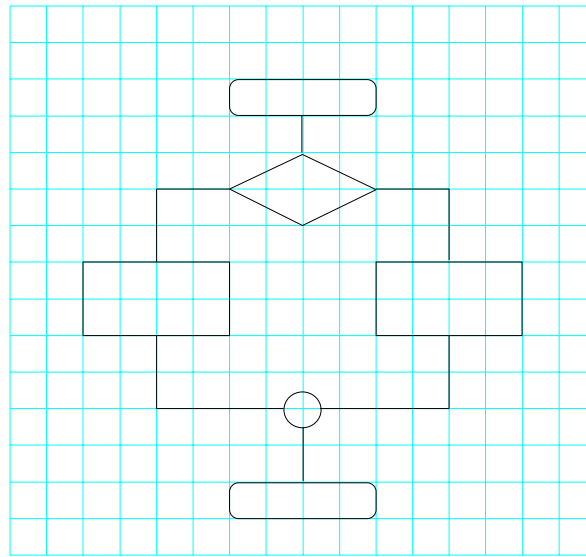


Figure VI-1: Using a Quadrille Pad for Flowcharting

You can get a 17" X 22" desk pad with pale blue, quarter-inch rulings. This gives you room to accommodate as many nestings and compoundings as your most complex module demands. You can fold a sheet of this paper twice to get an 8½" X 11" folio that's easy to carry around, if you're the kind of person who takes a lot of work home. If you reproduce it on an office photocopier, the blue lines usually don't show up, leaving bystanders to gawk at your drafting prowess. If this kind of thing is important to you, turn the paper over. You can see the lines well enough through this paper to draw the flowchart on the back. Your flowchart then looks truly impressive when you leave it lying casually on your desk.

You can also get some very nice engineering pads with eighth-inch quadrille rulings on one side and straight line rulings on the other. You can carry this around with you to scribble flowcharts and written notes into. You never know where you'll be when the spirit takes you.

For all this, try not to let people use your original flowcharts for documentation. Even freehanded in pencil, much erased, crudely annotated, and dog-eared from being hauled around in your briefcase, they look good. "Good" in this context means "authoritative." If

you make a mistake, and someone else has to correct it, that person won't touch your flowchart out of reverence for the authority it exudes. This really happens, and it's bad.

For original programs, you really don't need the flowchart: The psuedocode takes its place. I always throw the flowchart away as soon as I've coded an original program, not least because they look like that which ought to be thrown away after I've hacked them up.

For maintenance work, you can keep your flowcharts because you don't have time to psuedocode other peoples' programs. Existing work is more stable, hence safer to document with flowcharts. I still caution against your letting other people see them, though. If you make a mistake, the authoritative appearance of your flowchart can keep a more diffident soul from finding the problem.

Don't, incidentally, confuse program flowcharts as we use them here with system flowcharts that show the relationships of programs to the data they manipulate. Your shop standards will almost certainly demand that you submit system flowcharts as documentation. Those charts are a lot more stable, though, than charts of individual program structures. They're also much easier to change when your system does. Withal, they are useful. Do them exactly as your standards manual prescribes.

Pseudocoding Style

The trick of making psuedocode look good is to locate it close enough to your code line to keep code and psuedocode from constantly overlapping, yet far enough away to allow you a useful number of nesting levels. The best place to start psuedocode in a BASIC program seems to be around column 30, but it varies from program to program.

Don't use more than two or three spaces for indenting, or you'll quickly run out of room. Once you discover how easy it is, you find yourself nesting as deeply as ever you need

to to get the job done. Everyone I've seen using these methods has done just that, sometimes even to my astonishment.

Be careful not simply to echo in psuedocode what the code already says, as in the following example:

```
1000 A=B          ' NOW LET A EQUAL B
```

If this really worked, “self–documenting” programs in languages like COBOL and C would always be as easy as psuedocode to read and understand. Somewhere within your first five minutes as a maintenance programmer, you discover that no self–documenting program reads clearly, even when your predecessor has tried ever so hard to make it so.

The problem is that self–documenting code mires itself in detail. If you psuedocode, you can explain the most minute operation of your program in terms of the big ideas to which it's subordinate. This brings the full power of your brain to bear on draining the swamp in spite all those alligators. Your brain is much more powerful than any mainframe, after all. You abuse it when you try to make it operate at the level of a mere machine.

Compositional Style

A good, cheap trick of documenting programs is to make a typewriter cartoon of your functional hierarchy in a starred box near the beginning of your program. As a rule, shop standards require you to put a lot of other information in starred boxes at the beginning as well. If you include your functional hierarchy, it should follow the required boxes. I always code page breaks to put it on a separate page of my listing.

One major, purchased system I dealt extensively with required the functional hierarchy at the beginning of a program. It also required a second starred box containing the psuedocode for every module. Alas, this caused programmers to write the psuedocode first,

then write the module. Every time they changed their minds, the program code changed radically, but the psuedocode didn't. Writing the psuedocode as part of the instruction stream itself is the only way I know to keep it meaningful.

Beyond this, you don't really have many choices in composing programs. You can decide to use a subroutine or leave a complex structure nested under some other structure. You can decide how many statements should go into a module you describe in a barred box in your flowchart.

You can decide to put a loop body in a subroutine, leaving its control structure on the level above. If you do this, you'll avoid long loops and make your loop control structures really stand out.

You can put the whole loop in a subroutine, writing its control structure conditions in psuedocode beside the subroutine call. This makes your program look more like your flowchart.

You might use both techniques in the same program, depending on what you want to accomplish at a given moment. None of this can change your functional hierarchy. None of this can change the real composition of your program. Its only benefit is as documentation.

Structuralists early on noticed one problem in composition that many programmers encounter. That is the problem of changing variables in ways that have unexpected effects on other parts of your program.

This is a particularly irritating problem for maintenance programmers, because it's difficult to find out how an erratic variable originally got its bad value. Especially after midnight when your production system is down.

The structuralist answer to this was to build programming languages that keep subroutines from overwriting data from other subroutines. In effect, they make subroutines act as separate programs that you can call from your main program, changing only the variables you specifically pass to the subroutine. This general approach to composition, by the way, was responsible for the shop standards that produced the overlapping loop example of Chapter IV.

You can isolate variables by writing “external” subroutines: That is, subroutines that exist as separate programs on your disk. You can also do it in BASIC or FORTRAN by writing your subroutines as functions, which take only certain arguments. Of course, you can always pass the wrong arguments with the same, sad result in any language. You can also arrange things so that all your variables get passed to every subroutine in your program, nullifying the effect of using this technique in the first place. Never underestimate the perverse capacity of programmers you’ve insulted by making it “impossible” for them to cause you pain.

Just be careful to isolate variables to specific functions. You can do this with a series of heavily-commented LETs at the beginning of a BASIC program, but it usually isn’t necessary. The name of each variable can and should suggest the function that operates on it, wherever you define it.

Style and Optimization

The way you write a program can affect the way it runs. On my 80286-based PC, the second version of the example program of Appendix A runs faster than the first. Not much faster, mind you, but faster nevertheless. On an older computer, such as an Intel 8088 or a NEC V20, there would probably be a greater difference. On a 1-Mhz Commodore 64, the

difference might be more noticeable yet. On the slowest of these machines, the program would do the job several dozen times faster than I could do it in a word processor.

“Optimization” is always a compromise. The first question you have to ask is, “what resource do I want to optimize?” The last time I looked, an Intel i486-based motherboard cost around \$300. A gigabyte of magnetic disk storage went for around \$2500, and optical storage was quickly becoming cheaper than that. A megabyte of RAM would set you back thirty-eight bucks.

The lowliest maintenance programmer’s time costs between thirty-five and sixty dollars an hour, once you pay taxes and benefits, and the price goes up from there depending on the complexity of the work. It doesn’t take too many of these hours to justify buying additional memory and disk, or even a faster and more capable central processor.

Don’t be taken in, therefore, by people who tell you that the second example in Appendix A is more “efficient” than the first. That would only be true if the program never changed, or if it didn’t cost me anything to program.

If you’re worried about it, you can sometimes speed things up by the way you define variables. If you use a counter, for example, make it an integer variable. If you can, store numbers as binary integers rather than as text strings that have to be converted. Bear in mind, though, that even in the on-line environment, in which users enter information directly, updating the data base as they enter, the savings you get this way typically add up to a few milliseconds per transaction. It’s usually cheaper just to get a faster computer than it is to rewrite applications for efficiency.

Eclat

You can greatly enhance your standing and credibility among other programmers by referring to the computer or any of its programs as “he.” It’s also helpful to sprinkle your conversation liberally with references to “splat” for the asterisk, “bang” or “dammit” for the exclamation point, “hat” for the circumflex accent (^), and “pipe” for the vertical character (|). One programmer of my acquaintance memorialized himself by calling the splat an “asterkiss,” but that may be going too far.

You can distinguish yourself from your structuralist colleagues by showing respect for the minds and work of your fellow programmers. They’ve created a vast body of programming, and it all works. Avoid, therefore, criticizing another person’s program until you’re sure you know exactly what it does that you object to. Even then, couch your criticism in terms of what you’d have the programmer do. Keep in mind that your boss or your most important mentor may have written a program you ridicule.

Remember that anything worth doing is worth doing for kicks. Go out and start some fires.

Appendix A: The Example Program

Problem Description

This program reformats tables created from spreadsheet PRINT routines. Those routines produce columnar information, separating the columnar data with spaces. I want to set these tables in Ventura Publisher (or some other desktop publisher) without retyping them. Because these tables, moreover, contain extensive formatting, the table functions built into Ventura Publisher leave a lot of work to be done to retain that formatting. This program translates heavily formatted table files to tagged files in which the columnar data are separated by the tab [CHR\$(9)] character.

This version of the program uses the Ventura tag for “TAB,” which is <9>. That way, you can see what the output actually looks like.

Input Description

In my job, I don't tell a user what a table is supposed to look like. The user tells me. The decisions I make have to do with what font to use at what point size. Maybe I get to choose the paper I print it on. What I certainly don't determine is the formatting.

Users generally want to generate the table information in a spreadsheet. As a rule, they design their spreadsheets to group products in response to bid specifications. They have to provide subtotals for each group, and each group as a rule has to be preceded by a header. Often, they group the groups. Almost always, they want to put extended descriptive information in block paragraphs in the middle of the table somewhere.

Your garden variety spreadsheet usually has undesirable blank lines in it, often has header information of more than one kind spread randomly throughout it, and is otherwise composed of

tabulated information separated by spaces into columns. The main requirement is to remove the spaces and replace them with tabs depending on the width of the columns.

The garden-variety spreadsheet also contains a lot of other trash, like equal-sign filled cells, dash filled cells, page breaks, and so forth.

Input fields are either right justified or left justified. Right justified fields, if numeric, may or may not contain one blank to the right of the field for a sign.

Control Input

Because there are so many variations of the way a spreadsheet can be formatted, the program requires a control file to describe the specific print file to be converted. Since a spreadsheet can be offset from the top or right, the control file has to allow for removing leading spaces to the left or leading lines at the top.

The control file also has to provide a means of translating code characters into Ventura Publisher tags. These fall into two categories: Tags that format header information, and tags that format table lines. The input table file has to be edited to include these formatting code characters.

For the tabulating function to work, the control file has to provide the width and the justification, right or left, for each column in the input table, as well as the number of fields to be translated.

The file "QUOTE.PRT" is a print file from a spreadsheet I use that exhibits all of the characteristics I've found it necessary to deal with programmatically.

The spreadsheet on which I based QUOTE.PRT has had some reformatting done to it. If I'm working with a spreadsheet I have to set all the time, I can always talk my user into building the

spreadsheet this way: Such users won't present spreadsheets any other way once they've seen how Ventura makes them look.

If I'm working with casual users, I go into the ASCII print file and edit the information myself. I take out any symbol-filled cells, but leave whole lines of dashes and blank lines alone. The program gets those. The program also removes page breaks. If the page break occurs on an otherwise blank line, the program treats that like any blank line, which is to say, it eliminates it.

I'd also edit any centered information, justifying it either to the right or to the left in its fields. I've never been asked to do this, though. A typical spreadsheet user will always left-justify text information and its associated headers, and will always right-justify numerical information and its associated headers. If someone gives me something that's really off the wall, I usually get a copy of the original spreadsheet and modify that until it looks the way I want it to. If it's really, truly bizarre and the user insists on my retaining this, I set it by hand. Life is too short to try to figure out everything people do in a computer program, and I'd probably want to use a different tabulating method for stuff like that in any event.

Output Specification

The output file consists in tabular lines in which the columns are separated by tab symbols.

Input lines that begin with "@" are header lines. The program reads the @ symbol in the first column as an instruction to place whatever comes after it unaltered into the output file. If you want to put a tag in there explicitly, then, you can do so by typing "@@TAG =" or whatever at the beginning of the line. I know this is awkward, but I first wrote this program for the Aldus Pagemaker in my pre-enlightenment days. You get used to it.

If you follow the @ sign with any of five other symbols, the program builds a corresponding Ventura tag on that line. The program gets the tag names from a DATA statement in line 9999.

The symbols are +, \$, ^, #, and %. They're hard coded in the program, but you can change them, too, if you want to. CONTROL.FIL contains a reference list of the symbols and the tags that come with the program.

If you begin a line with the tilde (~) symbol, the program will tag that line as @TABLE HEAD = . If you start a line with a bang (!), the program tags the line as @TOTAL LINE = . If you don't put anything in the first column, the program assigns the @TABLE LINE = tag. You can then modify the TABLE.STY included in this package to make these tabbed lines look any way you want.

The spreadsheet is a template of what I actually use to make computer system quotes. It figures markups from costs, computes lease rates, and generally keeps things accurate.

Running the Example Program

To run the program against the sample data I've included here, type GWBASIC (or BASIC, or BASICA — whatever comes stock on your computer) followed by a space, followed by YAFVPTP. Every time the program processes a line, it prints a dot to the screen to indicate life.

To run the program against a spreadsheet print file of your own, you need to modify a file called CONTROL.FIL which must always be in the same directory as the one you run YAFVPTP from (unless, of course, you alter the BASIC program that hard codes it).

CONTROL.FIL tells YAFVPTP how many of your print file's fields you want to format for Ventura Publisher, together with a detailed description of each file you want to format. You can format as many files as you like with this program. You can format more than one file on the same pass. You can format multiple files with different characteristics in one pass.

If you want to format more than one spreadsheet on one pass, you have to enter an entire set of parameters for the second and each subsequent spreadsheet. The spreadsheets don't, therefore, have

Programming

to have the same characteristics. Don't separate the entries in CONTROL.FIL for multiple spreadsheets with blank lines, symbols, or suchlike.

CONTROL.FIL ends with an asterisk in the first column. It will also end on an end-of-file, but an awful lot of word processors/text editors will stick an extra carriage return at the end of the file. Besides, I stuff a lot of miscellaneous junk in that file after the asterisk. If you take out the asterisk and run the program anyway, you'll get the data just fine, but the program will terminate abnormally, leaving you in BASIC. Type SYSTEM to get out.

Control File Input

A typical CONTROL.FIL entry might look like this:

```
0
0
0
12R15L50L05R12R12R
QUOTE.PRT
QUOTE.TXT
9
0
0
12R15L50L05R12R12R12R12R12R
ENQUOTE.PRT
ENQUOTE.TXT
* = File terminator for the program: Don't touch the asterisk.
+ = Title
% = Elist
$ = Head Level 1
^ = Head Level 2
# = Page Break
```

This stuff is here for reference only: The program
doesn't use it. To change style tags that the
program assigns, modify the DATA statement in the
last program line. Symbols are hard-coded.

This is, in fact, the CONTROL.FIL of this appendix.

The first line is the number of fields you want to convert. Your print file has to have at least this many fields, but can have fewer if you like. Note that QUOTE.PRT indeed has more fields than I take out of there.

The second line is the vertical offset, and the third line is the horizontal offset. Most spreadsheets have some default printing options that cause them to pad the top and/or bottom with blank lines, and to pad the left margin with spaces. You can easily avoid generating files like this, but you may have to work with an unsophisticated user who will give you such a file 100 times out of 100. This is for that user.

The fourth line describes each line of the input file. It is a collection of three-character groups, one for each field you want to convert.

Programming

The first two characters of each group are numeric, and represent the width of the column in your input file. These widths are the same in the print file as they are in the original spreadsheet. You can ask your user to either give you a copy of the spreadsheet itself or to tell you what the column widths are when you get the print file.

The third character of each group is the justification, either right or left.

The fifth line is the name of the input file, and the sixth line is the name of the output file. These can be fully-qualified if you like, allowing you to run YAFVPTP from a directory other than the one in which the files reside. Remember that CONTROL.FIL has to be wherever you run YAFVPTP from unless you modify the program itself to direct YAFVPTP to look for it somewhere else. Remember too that life is short and disk is cheap. I usually copy YAFVPTP and CONTROL.FIL into the directory I'm using to make a quote. That way, if I change it, or if I change CONTROL.FIL, I can still modify or reproduce an old quote without messing with the program. Since I devote a directory to each proposal for ease of backup and filing, this works well for me.

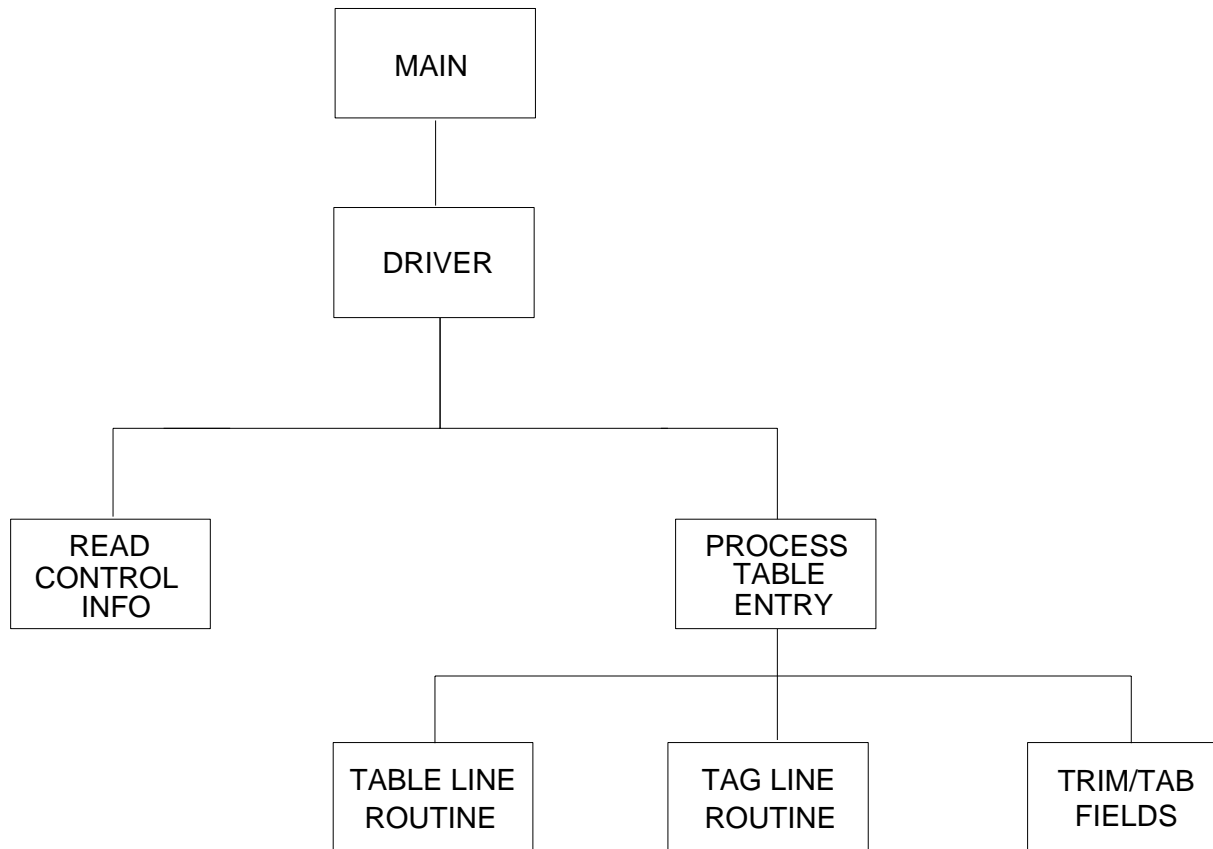
Notice that CONTROL.FIL in this example is set up to convert two different files. The asterisk in the first column in the line immediately following the second output file name is the file terminator. That allows me to stuff all kinds of documentary trivia into it, aimed at helping you set up or edit spreadsheets that you know you'll want to format this way. Notice also that the format for the two spreadsheet print files is the same, but that the outputs are different. You can format only the leftmost n columns of a file if you want to.

Miscellany

Don't worry about page breaks. The program eliminates them. If the page break is the CHR\$(12) (female symbol), the program takes it out. If the page break is just a bunch of blank lines,

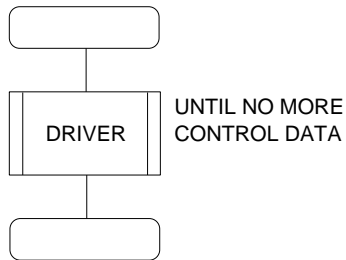
the program removes them, too. YAFVPTP takes out all blank lines. If you *want* a blank line in your output file, put an @ all by itself in the first column of your print file.

Functional Hierarchy

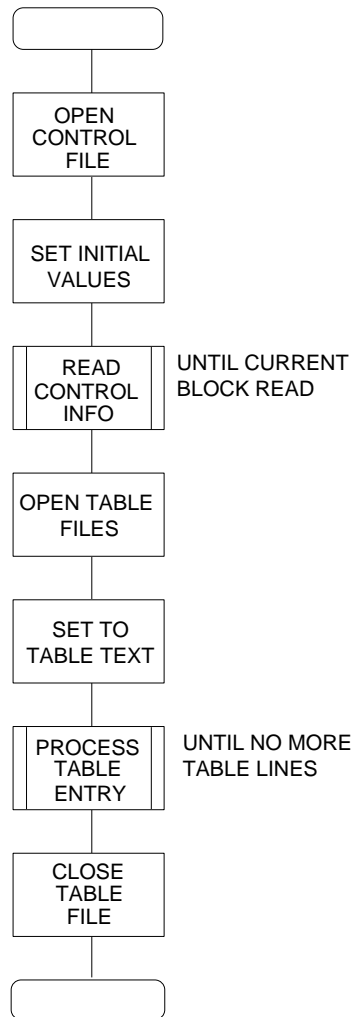


Program Flowchart

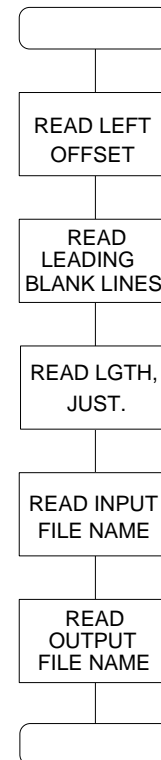
MAIN



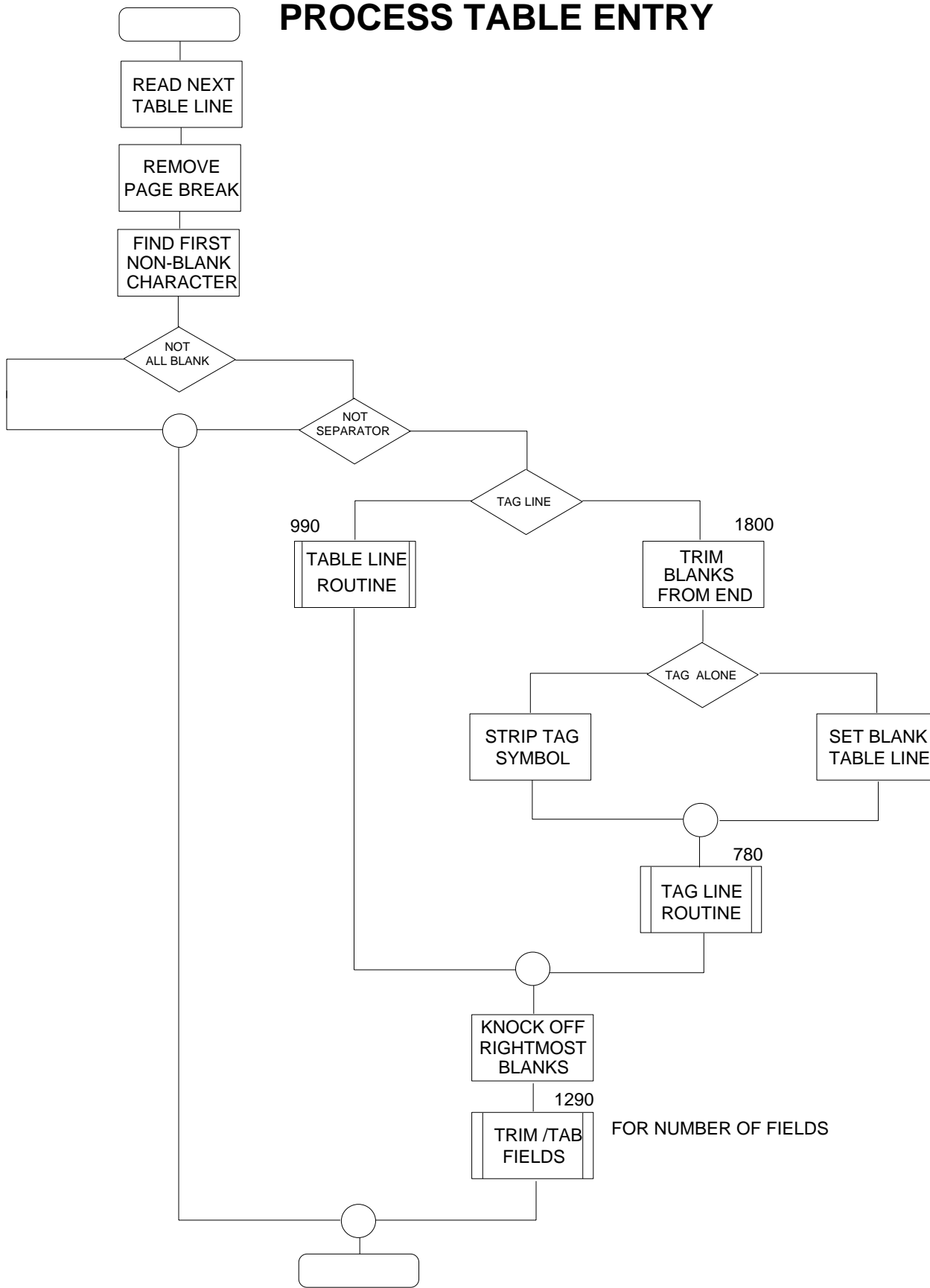
DRIVER

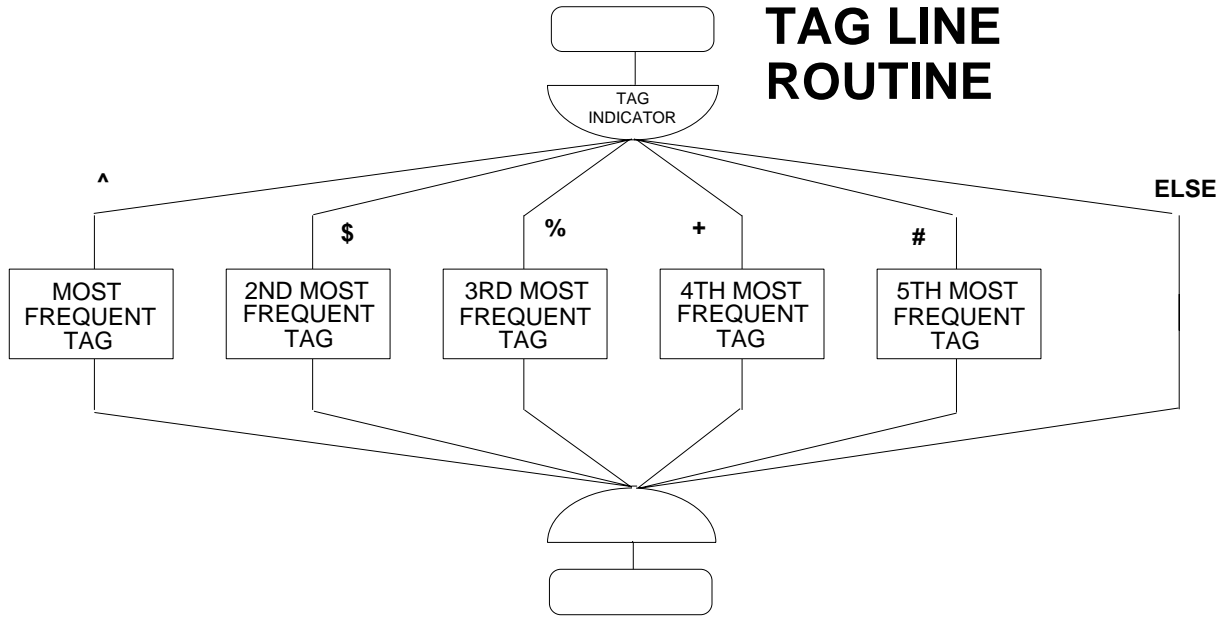


READ CONTROL INFO



PROCESS TABLE ENTRY





TRIM/TAB FIELDS

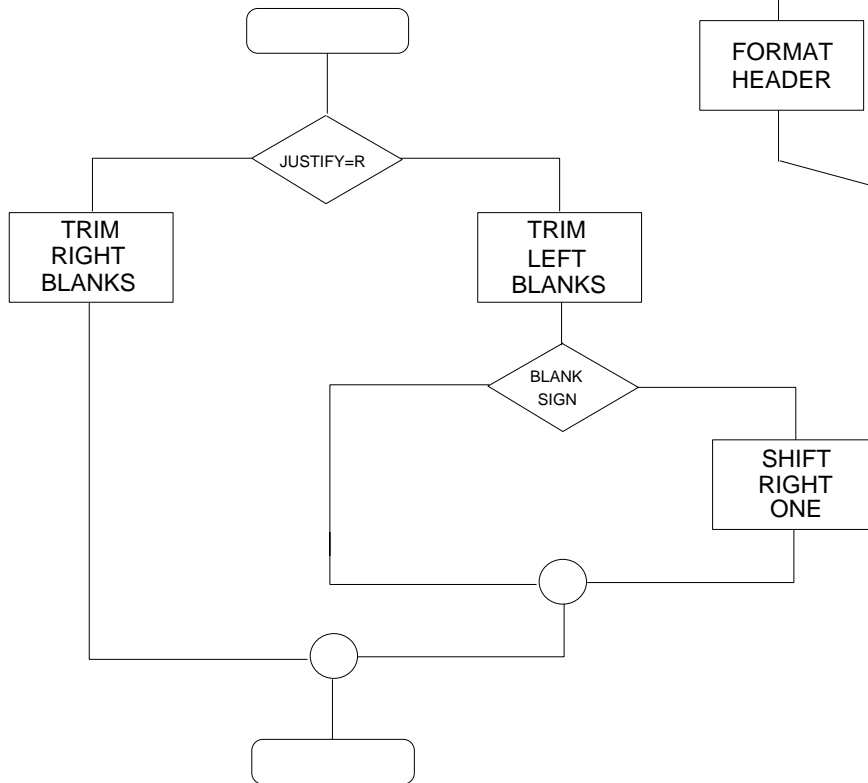
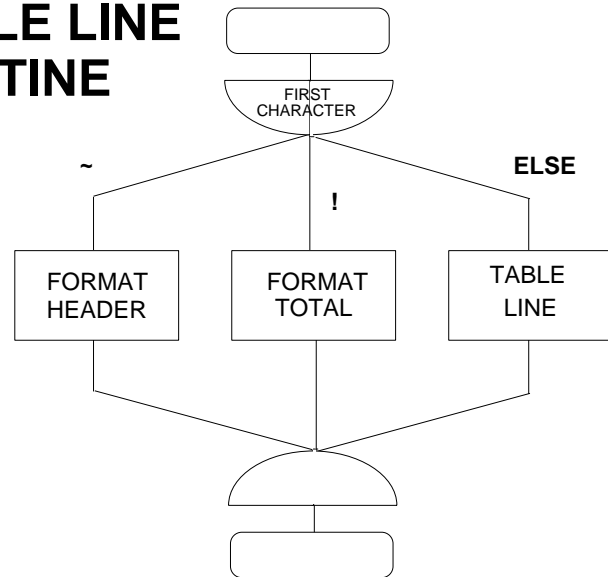


TABLE LINE ROUTINE



First Example Listing

```
10 REM *****
20 REM * YAFVPTP: A PROGRAM FOR TABULATING FORMATTED SPREADSHEETS WITH MANY *
30 REM * HEADERS AND THREE LINE STYLES. USEFUL IF YOU LIKE TO MAKE TABLES *
40 REM * WITH A LOT OF EXPLANATORY STUFF THAT SHOULDN'T BE FORMATTED LIKE *
50 REM * THE TABLE LINES. THIS IS THE FIRST OF TWO PROGRAM LISTINGS, BOTH *
60 REM * OF WHICH PERFORM THE SAME FUNCTION. THIS IS THE PROGRAM AS *
70 REM * ORIGINALLY WRITTEN AND MUCH REVISED, AS THE AUTHOR ADAPTED IT TO A *
80 REM * GREAT NUMBER OF DIFFERENT SPREADSHEET OUTPUTS. NICED UP VERSION. *
90 REM *****
100 ON ERROR GOTO 1750
110 DIM F$(60),JUSTIFY$(60),FLENGTH(60),FTYPE$(60)
120 REM *****MAIN PROGRAM AND INITIALIZATION ROUTINES*****
130 REM          OPEN THE CONTROL FILE
140 OPEN "CONTROL.FIL" FOR INPUT AS #1
150 TABX$="  " ' SET INITIAL VALUES FOR MAIN ROUTINES
160 READ TAG0$,TAG1$,TAG2$,TAG3$,TAG4$
170 REM *****MAIN PROGRAM LOOP*****
180 IF EOF(1) THEN 510 ' REPEAT WHILE MORE RECORDS IN CONTROL FILE
190 LINE INPUT #1, IMAX$ ' READ THE NUMBER OF FIELDS
200 IF LEFT$(IMAX$,1)="*" THEN 510
210 IMAX=VAL(IMAX$)
220 LINE INPUT #1, LOFFSET$ 'READ THE TABLE OFFSET TO THE LEFT
230 LOFFSET=VAL(LOFFSET$)
240 LINE INPUT #1, TOFFSET$ 'READ THE NUMBER OF BLANK LINES BEFORE THE TABLE
250 TOFFSET=VAL(TOFFSET$)
260 LINE INPUT #1, X$ ' READ THE FIELD LENGTH AND JUSTIFICATION LINE
270 K=1
280 FOR I=1 TO IMAX ' FOR THE NUMBER OF FIELDS IN THE TABLE
290 REM          GET THE FIELD LENGTH
300 FLENGTH(I)=VAL(MID$(X$,K,2))
310 K=K+2 ' RESET POINTER TO JUSTIFICATION CHARACTER
320 JUSTIFY$(I)=MID$(X$,K,1)
330 K=K+1 ' SET TO NEXT LENGTH AND JUSTIFICATION FIELD
340 NEXT I ' CONTINUE
345 REM          END REPEAT
350 LINE INPUT #1, IX$ ' READ THE INPUT FILE NAME
360 PRINT ""
370 PRINT IX$
380 LINE INPUT #1, OX$ ' READ THE OUTPUT FILE NAME
390 PRINT OX$
400 REM          ' OPEN THE INPUT AND OUTPUT FILES
410 OPEN IX$ FOR INPUT AS #2
420 OPEN OX$ FOR OUTPUT AS #3
430 IF TOFFSET=0 THEN 465
440 FOR I=0 TO TOFFSET ' FOR THE NUMBER OF BLANK LINES PRECEDING TABLE
450 LINE INPUT #2, X$ ' WASTE THEM
460 NEXT I ' CONTINUE
465 REM          END REPEAT
470 GOSUB 540 ' TAKE SPACES OUT OF THE TABLE FOR THIS ENTRY
480 CLOSE #2 ' CLOSE INPUT AND OUTPUT FILES
```

Programming

```
490 CLOSE #3
500 GOTO 180          ' CONTINUE
510 REM              END REPEAT
520 CLOSE #1        ' TERMINATE
530 SYSTEM
540 IF EOF(2) THEN 1690 ' REPEAT WHILE NOT END OF FILE
550 LOFFSET=VAL(LOFFSET$)
560 REM *****TABLE MANIPULATION ROUTINE*****
570 LINE INPUT #2, X$
573 REM              TAKE OUT PAGE BREAKS
575 IF LEFT$(X$,1)=CHR$(12) THEN X$=RIGHT$(X$,LEN(X$)-1)
580 K=LEN(X$)        ' FIND THE FIRST NONBLANK CHARACTER:
590 FOR I = 1 TO K    ' FOR INPUT STRING LENGTH OR UNTIL BLANK HIT
600 IF MID$(X$,I,1) <> " " THEN 620
610 NEXT I           ' CONTINUE
620 REM              END REPEAT
630 I=I-1
635 REM              ' IF LINE NOT ALL BLANK AND NOT SEPARATOR
640 IF LEN(X$)=0 THEN 1670
650 IF I = K THEN 1670
660 IF LEFT$(X$,10)="======" OR LEFT$(X$,10)="———" THEN 1660
670 PRINT ". ";
680 REM              IF THIS IS A TAGLINE
690 IF LEFT$(X$,1)<>"@" THEN 970
700 GOSUB 1800        ' TRIM THE BLANKS OFF THE END OF IT
710 IF X$<>"@" THEN 750 ' IF THE RESULT IS A SINGLE CHARACTER
720 X$="@TABLE LINE = " ' THIS IS A BLANK TABLE LINE
730 PRINT #3, ""
740 GOTO 770          ' ELSE
750 X$=RIGHT$(X$,LEN(X$)-1) ' GET RID OF THE TAG SYMBOL
760 PRINT #3, ""
770 REM              ' ENDIF
780 REM              ' BEGIN TAG ROUTINE CASE STRUCTURE
790 IF LEFT$(X$,1)<>"^" THEN 820 ' WHEN TAG INDICATOR IS "^"
800 PRINT #3, TAG4$;      ' SET MOST FREQUENTLY-USED TAG
810 GOTO 930              ' END CASE
820 IF LEFT$(X$,1)<>"$" THEN 850 ' WHEN TAG INDICATOR IS "$"
830 PRINT #3, TAG3$;      ' SET SECOND MOST FREQUENT TAG
840 GOTO 930              ' END CASE
850 IF LEFT$(X$,1)<>"%" THEN 880 ' WHEN TAG INDICATOR IS "%"
860 PRINT #3, TAG2$;      ' SET THIRD MOST FREQUENT TAG
870 GOTO 930              ' END CASE
880 IF LEFT$(X$,1)<>"+" THEN 910 ' WHEN TAG INDICATOR IS "+"
890 PRINT #3, TAG1$;      ' SET FOURTH MOST FREQUENT TAG
900 GOTO 930              ' END CASE
910 IF LEFT$(X$,1)<>"#" THEN 940 ' WHEN TAG INDICATOR IS "#"
920 PRINT #3, TAG0$;      ' SET MOST INFREQUENT TAG
930 X$=RIGHT$(X$,LEN(X$)-1) ' REMOVE THE TAG SYMBOL
940 PRINT #3, X$          ' ELSE PRINT THE TAG
950 REM              ' END TAG ROUTINE CASE STRUCTURE
960 GOTO 1670           ' ELSE
970 REM
980 PRINT #3, ""          ' TERMINATE THE LINE
990 REM              BEGIN TAB ROUTINE CASE STRUCTURE
1000 IF LEFT$(X$,1)="~" THEN 1020 'WHEN FIRST CHARACTER IS TILDE
```

Programming

```
1010 GOTO 1040
1020 PRINT #3,"@TABLE HEAD = "; ' PRINT TAG FOR TABLE HEAD
1030 GOTO 1100 ' END CASE
1040 IF LEFT$(X$,1)="!" THEN 1060 'WHEN FIRST CHARACTER IS BANG
1050 GOTO 1090
1060 PRINT #3,"@TOTAL LINE = "; ' PRINT TAG FOR TOTALS/SUBTOTALS
1070 X$="~"+RIGHT$(X$,LEN(X$)-1) ' MARK FIELD FOR CLIPPING
1080 GOTO 1100 ' END CASE
1090 PRINT #3,"@TABLE LINE = "; 'ELSE PRINT TAG FOR TABLE LINE
1100 REM ' END TAB ROUTINE CASE STRUCTURE
1110 FLENGTH(0)=0 ' SET THE FIRST FIELD LENGTH AT 0
1120 LOFFSET=LOFFSET+1 ' SET THE TABLE OFFSET TO LEFT
1130 FOR I=1 TO IMAX ' FOR THE NUMBER OF FIELDS
1140 LOFFSET=LOFFSET+FLENGTH(I-1) 'SET TO NEXT FIELD
1150 LGTH=FLENGTH(I) ' SET THE FIELD LENGTH
1160 REM ' GET THE FIELD INTO A VARIABLE
1170 F$(I)=MID$(X$,LOFFSET,LGTH)
1175 REM ' IF LEFTMOST CHARACTER IS TILDE THEN
1180 IF LEFT$(F$(I),1)<>"~" THEN 1195
1185 REM ' LOSE THE TILDE
1190 F$(I)=RIGHT$(F$(I),LEN(F$(I))-1)
1195 REM ' ENDIF
1200 NEXT I ' CONTINUE
1205 REM ' END REPEAT
1210 K=0 ' NOW KNOCK OFF THE RIGHTMOST BLANK FIELDS
1220 FOR I=IMAX TO 1 STEP -1 ' FOR THE NUMBER OF FIELDS THERE ARE
1230 REM ' DETERMINE WHICH IS THE FIRST NON-NULL
1240 IF LEN(F$(I)) > 0 AND F$(I)<>SPACE$(LEN(F$(I))) THEN 1270
1250 NEXT I ' CONTINUE
1260 REM ' END REPEAT
1270 ISAVE = IMAX ' SAVE THE ORIGINAL TABLE VALUE
1280 IMAX = I ' REDUCE THE NUMBER OF FIELDS (OMIT TABS)
1290 FOR I=1 TO IMAX ' FOR THE NUMBER OF FIELDS THERE ARE
1300 REM ' IF THE FIELD IS NOT EMPTY
1310 IF F$(I)=SPACE$(LEN(F$(I))) THEN 1570
1320 IF JUSTIFY$(I)="R" THEN 1410 ' IF FIELD IS LEFT JUSTIFIED
1330 REM *****BEGIN SPACE TRIMMING ROUTINES*****
1340 FOR J=1 TO LEN(F$(I)) ' UNTIL YOU HIT A NON-BLANK, TRIM
1350 IF MID$(F$(I),(1+LEN(F$(I))-J),1)<>" " THEN 1390
1360 NEXT J ' CONTINUE
1370 REM ' END REPEAT
1380 REM ' OUTPUT FIELD IS TRIMMED FIELD
1390 Y$=LEFT$(F$(I),(1+(LEN(F$(I))-J)))
1400 GOTO 1470 ' ELSE (I.E., FIELD IS RIGHT JUSTIFIED)
1410 FOR J=1 TO LEN(F$(I)) ' UNTIL YOU HIT A NON-BLANK, TRIM
1420 IF MID$(F$(I),J,1)<>" " THEN 1460
1430 NEXT J ' CONTINUE
1440 REM ' END REPEAT
1450 REM ' OUTPUT FIELD IS TRIMMED FIELD
1460 Y$=RIGHT$(F$(I),(1+LEN(F$(I))-J))
1470 REM ' ENDIF
1480 REM ' IF RIGHTMOST BYTE IS BLANK
1490 IF RIGHT$(Y$,1)=" " THEN 1520
1500 YY$=Y$ ' THIS STRING IS OK
1510 GOTO 1530 ' ELSE
```

Programming

```
1520 YY$=LEFT$(Y$,LEN(Y$)-1)      '      TRIM THE STRING
1530 REM                          '      ENDIF
1540 PRINT #3,YY$;                '      OUTPUT THIS MESS
1550 IF I<IMAX THEN PRINT#3,TABX$;
1560 GOTO 1620                    '      ELSE
1570 IF I<IMAX THEN 1600          '      IF THIS IS THE LAST FIELD THEN
1580 PRINT #3,""                 '      TERMINATE THE LINE
1590 GOTO 1610                    '      ELSE
1600 PRINT #3,TABX$;              '      SET A TAB BY ITSELF
1610 REM                          '      ENDIF
1620 REM                          '      ENDIF
1630 NEXT I                       '      CONTINUE
1640 IMAX = ISAVE                 '      END REPEAT
1650 PRINT #3,""                 '      TERMINATE THE LINE
1660 REM                          '      ENDIF
1670 REM                          '      ENDIF
1680 GOTO 540                     '      CONTINUE
1690 REM                          '      END REPEAT
1700 RETURN
1705 REM                          ***** NORMAL TERMINATION *****
1710 CLOSE 2
1720 CLOSE 3
1730 RETURN
1740 REM                          ***** DEATH AND HELL *****
1750 PRINT ERR,ERL
1760 CLOSE #3
1770 CLOSE #2
1780 INPUT"HIT ENTER TO CONTINUE";YN$
1790 RESUME 520
1800 REM                          ***** TRIM BLANKS OFF TAGLINES *****
1810 IF LEN(X$)<=1 OR RIGHT$(X$,1)<>" " THEN 1880
1820 FOR L = LEN(X$) TO 1 STEP -1
1830 IF MID$(X$,L,1)<>" " THEN 1850
1840 NEXT L
1850 WY$=LEFT$(X$,L)
1860 X$=WY$
1870 REM
1880 RETURN
1890 DATA "@PAGE BREAK = ", "@TITLE = ", "@ELIST = ", "@HEAD LEVEL 1 = ", "@HEAD
LEVEL 2 = "
```

Second Example Listing

```
10 REM *****
20 REM * YASVPTP: A PROGRAM FOR TABULATING FORMATTED SPREADSHEETS WITH MANY *
30 REM * HEADERS AND THREE LINE STYLES. USEFUL IF YOU LIKE TO MAKE TABLES *
40 REM * WITH A LOT OF EXPLANATORY STUFF THAT SHOULDN'T BE FORMATTED LIKE *
50 REM * THE TABLE LINES. THIS IS THE SECOND OF THREE PROGRAM LISTINGS, ALL *
60 REM * OF WHICH PERFORM THE SAME FUNCTION. THIS VERSION OF THE PROGRAM *
70 REM * IS ONE THAT MIGHT HAVE BEEN WRITTEN IN THE TYPICAL DATA PROCESSING *
80 REM * PROGRAMMING SHOP. THE PROGRAM IS STRUCTURALLY IDENTICAL TO YAFVPTP.*
90 REM *****
100 ON ERROR GOTO 1750
110 DIM F$(60),JUSTIFY$(60),FLENGTH(60),FTYPE$(60)
120 REM *****MAIN PROGRAM AND INITIALIZATION ROUTINES*****
130 REM          OPEN THE CONTROL FILE
140 OPEN "CONTROL.FIL" FOR INPUT AS #1
150 TABX$="  " ' SET INITIAL VALUES FOR MAIN ROUTINES
160 READ TAG0$,TAG1$,TAG2$,TAG3$,TAG4$
170 REM *****MAIN PROGRAM LOOP*****
180 IF EOF(1) THEN 520 ' READ IN THE CONTOL INFORMATION
190 LINE INPUT #1, IMAX$ ' READ THE NUMBER OF FIELDS
200 IF LEFT$(IMAX$,1)="*" THEN 520
210 IMAX=VAL(IMAX$)
220 LINE INPUT #1, LOFFSET$ 'READ THE TABLE OFFSET TO THE LEFT
230 LOFFSET=VAL(LOFFSET$)
240 LINE INPUT #1, TOFFSET$ 'READ THE NUMBER OF BLANK LINES BEFORE THE TABLE
250 TOFFSET=VAL(TOFFSET$)
260 LINE INPUT #1, X$ ' READ THE FIELD LENGTH AND JUSTIFICATION LINE
270 K=1
280 FOR I=1 TO IMAX ' GET LENGTHS AND JUSTIFICATIONS IN MEMORY
300 FLENGTH(I)=VAL(MID$(X$,K,2))
310 K=K+2
320 JUSTIFY$(I)=MID$(X$,K,1)
330 K=K+1
340 NEXT I
350 LINE INPUT #1, IX$ ' READ THE INPUT FILE NAME
360 PRINT ""
370 PRINT IX$
380 LINE INPUT #1, OX$ ' READ THE OUTPUT FILE NAME
390 PRINT OX$
410 OPEN IX$ FOR INPUT AS #2
420 OPEN OX$ FOR OUTPUT AS #3
430 IF TOFFSET=0 THEN 470
440 FOR I=0 TO TOFFSET ' TAKE OUT BLANK LINES AT THE BEGINNING
450 LINE INPUT #2, X$
460 NEXT I
470 GOSUB 540 ' TAKE SPACES OUT OF THE TABLE FOR THIS ENTRY
480 CLOSE #2
490 CLOSE #3
500 GOTO 180
520 CLOSE #1 ' END THE PROGRAM
530 SYSTEM
```

Programming

```
540 IF EOF(2) THEN 1700 ' NOW READ IN THE INPUT FILE AND TRANSLATE IT
550 LOFFSET=VAL(LOFFSET$)
560 REM *****TABLE MANIPULATION ROUTINE*****
570 LINE INPUT #2, X$
575 IF LEFT$(X$,1)=CHR$(12) THEN X$=RIGHT$(X$,LEN(X$)-1)
580 K=LEN(X$) ' FIND THE FIRST NONBLANK CHARACTER
590 FOR I = 1 TO K
600 IF MID$(X$,I,1) <> " " THEN 630
610 NEXT I
630 I=I-1
640 IF LEN(X$)=0 THEN 540 ' IF ALL BLANKS THEN SKIP IT
650 IF I = K THEN 540
660 IF LEFT$(X$,10)="======" OR LEFT$(X$,10)="———" THEN 1670
670 PRINT ".";
690 IF LEFT$(X$,1)<>"@" THEN 980 '
700 GOSUB 1800 ' TRIM THE BLANKS OFF THE END OF IT
710 IF X$<>"@" THEN 750 ' IS RESULT MORE THAN ONE CHARACTER
720 X$="@TABLE LINE = " ' NO - THIS IS A BLANK TABLE LINE
730 PRINT #3,""
740 GOTO 790
750 X$=RIGHT$(X$,LEN(X$)-1) 'GET RID OF THE TAG SYMBOL
760 PRINT #3, ""
790 IF LEFT$(X$,1)<>"^" THEN 820 ' FIND OUT IF THIS IS A TAG LINE AND SET TAG
800 PRINT #3, TAG4$;
810 GOTO 930
820 IF LEFT$(X$,1)<>"$" THEN 850
830 PRINT #3, TAG3$;
840 GOTO 930
850 IF LEFT$(X$,1)<>"%" THEN 880
860 PRINT #3, TAG2$;
870 GOTO 930
880 IF LEFT$(X$,1)<>"+" THEN 910
890 PRINT #3, TAG1$;
900 GOTO 930
910 IF LEFT$(X$,1)<>"#" THEN 940
920 PRINT #3, TAG0$;
930 X$=RIGHT$(X$,LEN(X$)-1) 'REMOVE THE TAG SYMBOL
940 PRINT #3, X$ ' PRINT THE TAG
950 REM
960 GOTO 540 ' GO GET THE NEXT RECORD
970 REM
980 PRINT #3, "" ' TERMINATE THE LINE
1000 IF LEFT$(X$,1)="~" THEN 1020 'NOW FIGURE OUT IF THIS IS FORMATTED LINE
1010 GOTO 1040
1020 PRINT #3,"@TABLE HEAD = ";
1030 GOTO 1100
1040 IF LEFT$(X$,1)="!" THEN 1060
1050 GOTO 1090
1060 PRINT #3,"@TOTAL LINE = ";
1070 X$="~"+RIGHT$(X$,LEN(X$)-1)
1080 GOTO 1110
1090 PRINT #3,"@TABLE LINE = "; 'PRINT TAG FOR TABLE LINE
1110 FLENGTH(0)=0 ' SET THE FIRST FIELD LENGTH AT 0
1120 LOFFSET=LOFFSET+1 ' SET THE TABLE OFFSET TO LEFT
1130 FOR I=1 TO IMAX ' NOW GET THE FIELD LENGTHS INTO VARIABLES
```

Programming

```
1140 LOFFSET=LOFFSET+FLENGTH(I-1)
1150 LGTH=FLENGTH(I)
1170 F$(I)=MID$(X$,LOFFSET,LGTH)
1180 IF LEFT$(F$(I),1)<>"~" THEN 1200
1190 F$(I)=RIGHT$(F$(I),LEN(F$(I))-1)
1200 NEXT I
1210 K=0 ' NOW KNOCK OFF THE RIGHTMOST BLANK FIELDS
1220 FOR I=IMAX TO 1 STEP -1
1240 IF LEN(F$(I)) > 0 AND F$(I)<>SPACE$(LEN(F$(I))) THEN 1270
1250 NEXT I
1270 ISAVE = IMAX ' SAVE THE ORIGINAL TABLE VALUE
1280 IMAX = I ' REDUCE THE NUMBER OF FIELDS (OMIT TABS)
1290 FOR I=1 TO IMAX ' TRIM SPACES
1310 IF F$(I)=SPACE$(LEN(F$(I))) THEN 1570
1320 IF JUSTIFY$(I)="R" THEN 1410 'IS FIELD RIGHT JUSTIFIED?
1330 REM *****BEGIN SPACE TRIMMING ROUTINES*****
1340 FOR J=1 TO LEN(F$(I)) 'NO - TRIM SPACES FROM THE RIGHT
1350 IF MID$(F$(I),(1+LEN(F$(I))-J),1)<>" " THEN 1390
1360 NEXT J
1390 Y$=LEFT$(F$(I),(1+(LEN(F$(I))-J)))
1400 GOTO 1490
1410 FOR J=1 TO LEN(F$(I)) ' UNTIL YOU HIT A NON-BLANK, TRIM
1420 IF MID$(F$(I),J,1)<>" " THEN 1460
1430 NEXT J
1460 Y$=RIGHT$(F$(I),(1+LEN(F$(I))-J))
1490 IF RIGHT$(Y$,1)=" " THEN 1520 'BLANK SIGN BYTE?
1500 YY$=Y$ ' NO-THIS STRING IS OK
1510 GOTO 1540
1520 YY$=LEFT$(Y$,LEN(Y$)-1) ' TRIM THE STRING
1540 PRINT #3,YY$; ' OUTPUT THIS MESS
1550 IF I<IMAX THEN PRINT#3,TABX$;
1560 GOTO 1630
1570 IF I<IMAX THEN 1600 ' IS THIS NOT THE LAST FIELD?
1580 PRINT #3,"" ' NO - TERMINATE THE LINE
1590 GOTO 1630
1600 PRINT #3,TABX$; ' YES - SET A TAB BY ITSELF
1630 NEXT I
1640 IMAX = ISAVE
1650 PRINT #3,"" ' TERMINATE THE LINE
1680 GOTO 540 ' GO GET THE NEXT RECORD
1700 RETURN ' END OF SUBROUTINE
1710 CLOSE 2
1720 CLOSE 3
1730 RETURN
1740 REM ***** DEATH AND HELL *****
1750 PRINT ERR,ERL
1760 CLOSE #3
1770 CLOSE #2
1780 INPUT"HIT ENTER TO CONTINUE";YN$
1790 RESUME 520
1800 REM ***** TRIM BLANKS OFF TAGLINES *****
1810 IF LEN(X$)<=1 OR RIGHT$(X$,1)<>" " THEN 1880
1820 FOR L = LEN(X$) TO 1 STEP -1
1830 IF MID$(X$,L,1)<>" " THEN 1850
1840 NEXT L
```

Programming

```
1850 WY$=LEFT$(X$,L)
1860 X$=WY$
1870 REM
1880 RETURN
1890 DATA "@PAGE BREAK = ", "@TITLE = ", "@ELIST = ", "@HEAD LEVEL 1 = ", "@HEAD
LEVEL 2 = "
```

Sample Input

@\$E-4B SIL Computer Network (Unclassified)

@^Central Computer

1.	70069651	ALR MultiAccess 3000 MP, 486/33, 1 processor 16Mb RAM expandable to 64Mb, 8Kb int. cache 256Kb cache, 13 32-bit, C-Bus/EISA expansion slots 32-bit SCSI busmaster, 9 serial, 1 parallel port 17" color VGA 1024x768, .28dp, 1.2Mb 1.44Mb FDD ALR SuperVGA 16-bit high-performance VGA card Single 650Mb System Disk Drive, EISA I/O bus Tower chassis, 3 half-height internal drive bays	1	27,221	27,221	3,049	3,049
----	----------	---	---	--------	--------	-------	-------

@^Peripheral Equipment

2.	41200N	Seagate 1.05Gb Wren 5.25" f/h SCSI <15ms	1	2,620	2,620	293	293
3.	MER1	Maxtor 1GB Tahiti Microtower 1	1	4,423	4,423	495	495
4.	925045-00	Logitech Mouseman bus w/Windows 3.0	1	175	175	20	20
5.	OMNI-1350	Tripp Lite 1350VA/1200 WATTS UPS	1	879	879	98	98
6.	75-662	Tripp-Lite PowerMon Software UNIX	1	156	156	18	18
7.	PRO120	Proteon 120 32-bit EISA ETHERNET adapter	1	1,545	1,545	173	173
8.	4440	Genicom 800lpm draft, 165lpm NLQ DM printer Parallel and serial, bottom load paper path dual tractors	1	7,236	7,236	810	810

@^System Software and Services

9.	SA012-UX77	SCO UNIX SYSTEM V/386 Release 3.2	1	837	837	164	164
10.	SA014-UX77	SCO UNIX Development System 386 GT	1	837	837	164	164
11.	SA024-UX78	SCO UNIX/wMPX extension	1	837	837	164	164
12.	SA125-XG37	SCO VP/ix 386 2-users 5.25" AT	1	751	751	147	147
13.	SB128-UX77	TCP/IP Runtime for UNIX	1	596	596	117	117
14.	SA026-UX77	SCO UNIX NFS v. 1.1.1	1	500	500	98	98
15.	NVL893	Novell NETWARE 3.11 20-user	1	2,349	2,349	460	460
16.	NVL710	Novell NETWARE NFS 5.25 v 1.1	1	3,282	3,282	643	643
17.	883001408	NETWARE for MacIntosh 20 user	1	776	776	152	152
18.	OSI/SVPK	Installation, cabling, and connectors			1,625		

@%

!	Total			56,646		7,067	
---	-------	--	--	--------	--	-------	--

Programming

@%

@%

!	LEASE RATES	60 Mos.	48 Mos.
!		1,314	1,538

@%

!		36 Mos.	24 Mos.
!		1,891	2,775

@

@\$Options

1.	Substitute 1340Mb RAID Level 5 Disk Array For 1GB single system disk	9,104
----	---	-------

@%

2.	Add 670Mb expansion module to RAID array (Total expandability is to 16.75GBR)	3,977
----	--	-------

@%

3.	Add FlexStor Gigabyte Minicomputer Cabinet Includes 1 330Mb drive, power supply Assumes base unit with 330Mb vice 650Mb	4,654
----	---	-------

@%

4.	Add 33-MHz i486 SIO Card	6,982
----	--------------------------	-------

@%

5.	Add 33-Mhz i486 System Card	4,190
----	-----------------------------	-------

Sample Output

```
@HEAD LEVEL 1 = E-4B SIL Computer Network (Unclassified)
@HEAD LEVEL 1 = E-4B SIL Computer Network (Unclassified)
@HEAD LEVEL 2 = Central Computer
@HEAD LEVEL 2 = Central Computer
@TABLE LINE = 1.<9>70069651<9>ALR MultiAccess 3000 MP, 486/33, 1 processor
@TABLE LINE = 1.<9>70069651<9>ALR MultiAccess 3000 MP, 486/33, 1 processor
@TABLE LINE = <9><9>16Mb RAM expandable to 64Mb, 8Kb int. memory cache
@TABLE LINE = <9><9>16Mb RAM expandable to 64Mb, 8Kb int. memory cache
@TABLE LINE = <9><9>256Kb cache, 13 32-bit, C-Bus/EISA expansion slots
@TABLE LINE = <9><9>256Kb cache, 13 32-bit, C-Bus/EISA expansion slots
@TABLE LINE = <9><9>32-bit SCSI busmaster, 9 serial, 1 parallel port
@TABLE LINE = <9><9>32-bit SCSI busmaster, 9 serial, 1 parallel port
@TABLE LINE = <9><9>17" color VGA 1024x768, .28dp, 1.2Mb and 1.44Mb FD
@TABLE LINE = <9><9>17" color VGA 1024x768, .28dp, 1.2Mb and 1.44Mb FD
@TABLE LINE = <9><9>ALR SuperVGA 16-bit high-performance VGA card
@TABLE LINE = <9><9>ALR SuperVGA 16-bit high-performance VGA card
@TABLE LINE = <9><9>Single 650Mb System Disk Drive, EISA I/O bus
@TABLE LINE = <9><9>Single 650Mb System Disk Drive, EISA I/O bus
@TABLE LINE = <9><9>Tower chassis, 3 half-height internal drive
bays<9>1<9>27,221<9>27,221<9>3,049<9>3,049<9>n/a
@TABLE LINE = <9><9>Tower chassis, 3 half-height internal drive
bays<9>1<9>27,221<9>27,221<9>3,049<9>3,049<9>n/a
@HEAD LEVEL 2 = Peripheral Equipment
@HEAD LEVEL 2 = Peripheral Equipment
@TABLE LINE = 2.<9>41200N<9>Seagate 1.05Gb Wren 5.25" f/h SCSI
<15ms<9>1<9>2,620<9>2,620<9>293<9>293<9>n/a
@TABLE LINE = 2.<9>41200N<9>Seagate 1.05Gb Wren 5.25" f/h SCSI
<15ms<9>1<9>2,620<9>2,620<9>293<9>293<9>n/a
```

Programming

@TABLE LINE = 3.<9>MICROTOWER1<9>Maxtor 1GB Tahiti Microtower
1<9>1<9>4,423<9>4,423<9>495<9>495<9>n/a

@TABLE LINE = 3.<9>MICROTOWER1<9>Maxtor 1GB Tahiti Microtower
1<9>1<9>4,423<9>4,423<9>495<9>495<9>n/a

@TABLE LINE = 4.<9>925045-00<9>Logitech Mouseman bus w/Windows
3.0<9>1<9>175<9>175<9>20<9>20<9>n/a

@TABLE LINE = 4.<9>925045-00<9>Logitech Mouseman bus w/Windows
3.0<9>1<9>175<9>175<9>20<9>20<9>n/a

@TABLE LINE = 5.<9>OMNI-1350LAN<9>Tripp Lite 1350VA/1200 WATTS
UPS<9>1<9>879<9>879<9>98<9>98<9>n/a

@TABLE LINE = 5.<9>OMNI-1350LAN<9>Tripp Lite 1350VA/1200 WATTS
UPS<9>1<9>879<9>879<9>98<9>98<9>n/a

@TABLE LINE = 6.<9>75-662<9>Tripp-Lite PowerMon Software
UNIX<9>1<9>156<9>156<9>18<9>18<9>n/a

@TABLE LINE = 6.<9>75-662<9>Tripp-Lite PowerMon Software
UNIX<9>1<9>156<9>156<9>18<9>18<9>n/a

@TABLE LINE = 7.<9>PRO120<9>Proteon 120 32-bit EISA ETHERNET
adapter<9>1<9>1,545<9>1,545<9>173<9>173<9>n/a

@TABLE LINE = 7.<9>PRO120<9>Proteon 120 32-bit EISA ETHERNET
adapter<9>1<9>1,545<9>1,545<9>173<9>173<9>n/a

@TABLE LINE = 8.<9>4440<9>Genicom 800lpm draft, 165lpm NLQ DM printer

@TABLE LINE = 8.<9>4440<9>Genicom 800lpm draft, 165lpm NLQ DM printer

@TABLE LINE = <9><9>Parallel and serial, bottom load paper path

@TABLE LINE = <9><9>Parallel and serial, bottom load paper path

@TABLE LINE = <9><9>dual tractors<9>1<9>7,236<9>7,236<9>810<9>810<9>n/a

@TABLE LINE = <9><9>dual tractors<9>1<9>7,236<9>7,236<9>810<9>810<9>n/a

@HEAD LEVEL 2 = System Software and Services

@HEAD LEVEL 2 = System Software and Services

@TABLE LINE = 9.<9>SA012-UX77<9>SCO UNIX SYSTEM V/386 Release
3.2<9>1<9>837<9>837<9>164<9>164<9>n/a

@TABLE LINE = 9.<9>SA012-UX77<9>SCO UNIX SYSTEM V/386 Release
3.2<9>1<9>837<9>837<9>164<9>164<9>n/a

@TABLE LINE = 10.<9>SA014-UX77<9>SCO UNIX Development System 386
GT<9>1<9>837<9>837<9>164<9>164<9>n/a

Programming

@TABLE LINE = 10.<9>SA014-UX77<9>SCO UNIX Development System 386
GT<9>1<9>837<9>837<9>164<9>164<9>n/a

@TABLE LINE = 11.<9>SA024-UX78<9>SCO UNIX/wMPX
extension<9>1<9>837<9>837<9>164<9>164<9>n/a

@TABLE LINE = 11.<9>SA024-UX78<9>SCO UNIX/wMPX
extension<9>1<9>837<9>837<9>164<9>164<9>n/a

@TABLE LINE = 12.<9>SA125-XG37<9>SCO VP/ix 386 2-users 5.25"
AT<9>1<9>751<9>751<9>147<9>147<9>n/a

@TABLE LINE = 12.<9>SA125-XG37<9>SCO VP/ix 386 2-users 5.25"
AT<9>1<9>751<9>751<9>147<9>147<9>n/a

@TABLE LINE = 13.<9>SB128-UX77<9>TCP/IP Runtime for UNIX
5.25"<9>1<9>596<9>596<9>117<9>117<9>n/a

@TABLE LINE = 13.<9>SB128-UX77<9>TCP/IP Runtime for UNIX
5.25"<9>1<9>596<9>596<9>117<9>117<9>n/a

@TABLE LINE = 14.<9>SA026-UX77<9>SCO UNIX NFS v.
1.1.1/5.25"<9>1<9>500<9>500<9>98<9>98<9>n/a

@TABLE LINE = 14.<9>SA026-UX77<9>SCO UNIX NFS v.
1.1.1/5.25"<9>1<9>500<9>500<9>98<9>98<9>n/a

@TABLE LINE = 15.<9>NVL893/wC#VT605<9>Novell NETWARE 3.11 20-user
5.25"<9>1<9>2,349<9>2,349<9>460<9>460<9>n/a

@TABLE LINE = 15.<9>NVL893/wC#VT605<9>Novell NETWARE 3.11 20-user
5.25"<9>1<9>2,349<9>2,349<9>460<9>460<9>n/a

@TABLE LINE = 16.<9>NVL710/wC#VT604<9>Novell NETWARE NFS 5.25"/3.5" v
1.1<9>1<9>3,282<9>3,282<9>643<9>643<9>n/a

@TABLE LINE = 16.<9>NVL710/wC#VT604<9>Novell NETWARE NFS 5.25"/3.5" v
1.1<9>1<9>3,282<9>3,282<9>643<9>643<9>n/a

@TABLE LINE = 17.<9>883001408-002<9>NETWARE for MacIntosh 20 user
5.25"<9>1<9>776<9>776<9>152<9>152<9>n/a

@TABLE LINE = 17.<9>883001408-002<9>NETWARE for MacIntosh 20 user
5.25"<9>1<9>776<9>776<9>152<9>152<9>n/a

@TABLE LINE = 18.<9>OSI/SVPK-01<9>Hardware installation, cabling, and
connectors<9><9><9>1,625

@TABLE LINE = 18.<9>OSI/SVPK-01<9>Hardware installation, cabling, and
connectors<9><9><9>1,625

@ELIST =

@ELIST =

Programming

```
@TOTAL LINE = <9><9>Total<9><9><9>56,646<9><9>7,067
@TOTAL LINE = <9><9>Total<9><9><9>56,646<9><9>7,067
@TABLE LINE =
@TABLE LINE =
@TABLE LINE =
@TABLE LINE =
@TOTAL LINE = <9><9>LEASE RATES<9><9>60 Mos.<9>48 Mos.
@TOTAL LINE = <9><9>LEASE RATES<9><9>60 Mos.<9>48 Mos.
@TOTAL LINE = <9><9><9><9>1,314<9>1,538
@TOTAL LINE = <9><9><9><9>1,314<9>1,538
@TABLE LINE =
@TABLE LINE =
@TOTAL LINE = <9><9><9><9>36 Mos.<9>24 Mos.
@TOTAL LINE = <9><9><9><9>36 Mos.<9>24 Mos.
@TOTAL LINE = <9><9><9><9>1,891<9>2,775
@TOTAL LINE = <9><9><9><9>1,891<9>2,775
@TABLE LINE =
@TABLE LINE =
@HEAD LEVEL 1 = Options
@HEAD LEVEL 1 = Options
@TABLE LINE = 1.<9><9>Substitute 1340Mb RAID Level 5 Disk Array
@TABLE LINE = 1.<9><9>Substitute 1340Mb RAID Level 5 Disk Array
@TABLE LINE = <9><9>For 1GB single system disk<9><9><9>9,104
@TABLE LINE = <9><9>For 1GB single system disk<9><9><9>9,104
@ELIST =
@ELIST =
@TABLE LINE = 2.<9><9>Add 670Mb expansion module to RAID array
```

Programming

```
@TABLE LINE = 2.<9><9>Add 670Mb expansion module to RAID array
@TABLE LINE = <9><9>(Total expandability is to 16.75GBR)<9><9><9>3,977
@TABLE LINE = <9><9>(Total expandability is to 16.75GBR)<9><9><9>3,977
@ELIST =
@ELIST =
@TABLE LINE = 3.<9><9>Add FlexStor Gigabyte Minicomputer Cabinet
@TABLE LINE = 3.<9><9>Add FlexStor Gigabyte Minicomputer Cabinet
@TABLE LINE = <9><9>Includes 1 330Mb drive, power supply
@TABLE LINE = <9><9>Includes 1 330Mb drive, power supply
@TABLE LINE = <9><9>Assumes base unit with 330Mb vice 650Mb<9><9><9>4,654
@TABLE LINE = <9><9>Assumes base unit with 330Mb vice 650Mb<9><9><9>4,654
@ELIST =
@ELIST =
@TABLE LINE = 4.<9><9>Add 33-MHz i486 SIO Card<9><9><9>6,982
@TABLE LINE = 4.<9><9>Add 33-MHz i486 SIO Card<9><9><9>6,982
@ELIST =
@ELIST =
@TABLE LINE = 5.<9><9>Add 33-Mhz i486 System Card<9><9><9>4,190
```

Appendix B: Notes on Assembler Language and COBOL

The programming examples of this book are all in BASIC because more people have ready access to that language than to any other. A few things that programmers are called on to do all the time don't show up well in BASIC. A brief study of the two most popular data processing languages should clear up most potential confusion.

The IBM 360/370 Assembler Language

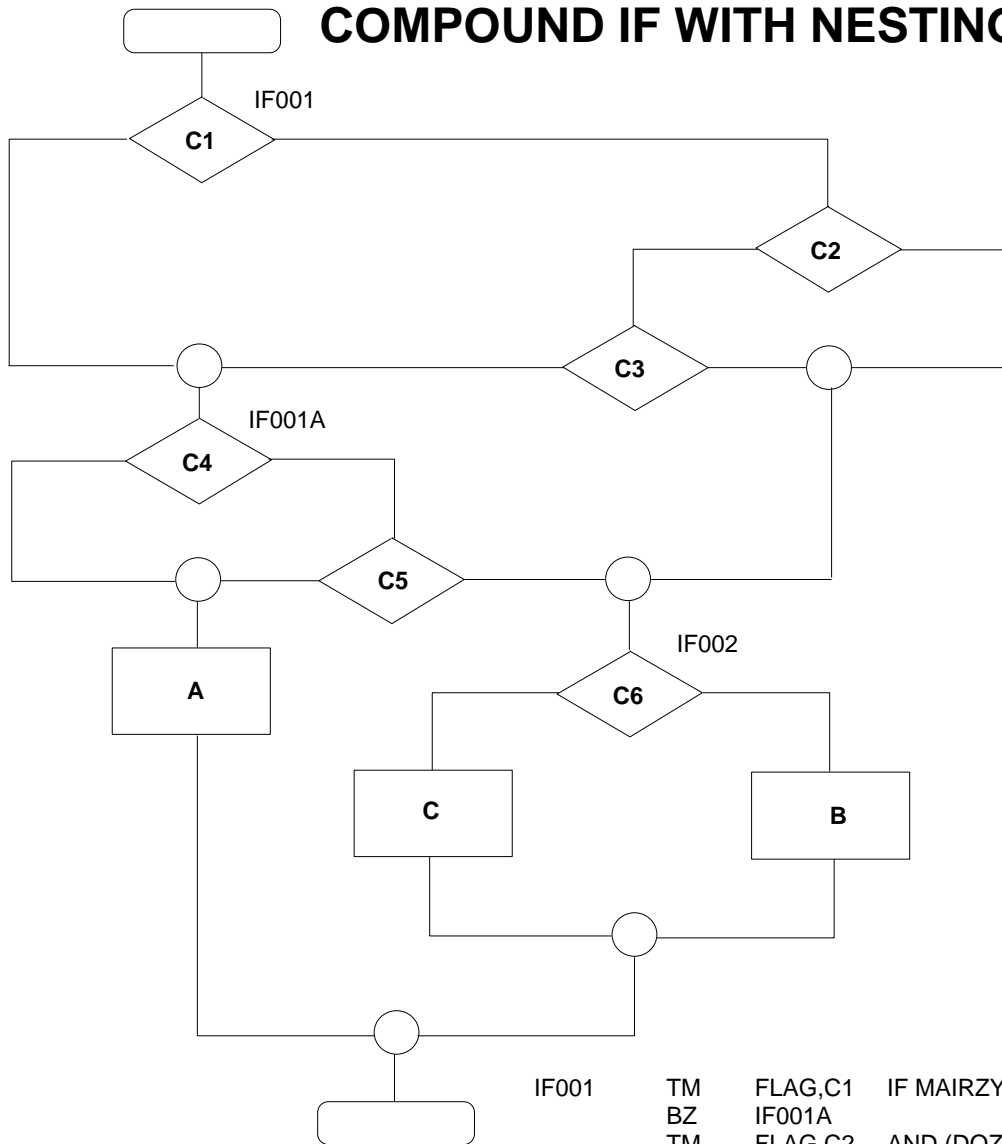
IBM's 360/370 assembler language is useful as an illustration of the way the methods work with languages that use named branch locations rather than line numbers. There are many such languages, more than a few of which are special implementations of BASIC.

Assembler syntax is somewhat more cryptic than BASIC's, but you can still program the most complicated examples you can imagine with only conditional branches and the unconditional branch (GOTO). Assembler lends itself nicely to this kind of coding, moreover, because the instruction itself is easy to code. If you use a named branch location, then you have to put that name in column one. The other parts of an assembler instruction can be anywhere within the eighty-character instruction line.

The operation code (verb), operands, and comments are separated only by spaces. You don't have to mess with the single quote to set off a comment as you do in BASIC.

The example of Figure B-1 is an assembler-language version of Figure III-11. I chose this figure because it's the most complicated structure in this book, and I wanted to show how easy it is to code such a structure even in assembler language. Notice that the only difference in the two flowcharts is that the BASIC line numbers are missing from the process boxes.

COMPOUND IF WITH NESTING



IF001	TM	FLAG,C1	IF MAIRZY DOATS
	BZ	IF001A	
	TM	FLAG,C2	AND (DOZY DOATS
	BO	THEN001	
	TM	FLAG,C3	OR LITTLE LAMZIE DIVIE)
	BO	THEN001	
IF001A	EQU	*	
	TM	FLAG,C4	OR (A KIDDLY DIVIE
	BZ	ELSE001	
	TM	FLAG,C5	AND YOU WOULD TOO)
	BZ	ELSE001	THEN
THEN001	EQU	*	
IF002	TM	FLAG,C6	IF ROOSTER LAYS DOORKNOB
	BZ	ELSE002	THEN
	BAL	12,B	OPEN A HARDWARE STORE
	B	ENDIF002	ELSE
ELSE002	EQU	*	
	BAL	12,C	GIVE UP
ENDIF002	EQU	*	ENDIF
	B	ENDIF001	ELSE
	BAL	12,A	EAT GRASS
ENDIF001	EQU	*	ENDIF

Programming

You can give a name to any location in your assembler program by coding an equate (EQU). When the operand is an asterisk, you're telling the assembler that the location you want to name is the location of the name itself.

The example code sets conditions by means of the "test under mask" (TM) instruction. For our purposes, this instruction checks a single bit in a single byte of storage. If it's a one, then the condition code is set to one. If it's a zero, the condition code is set to zero.

The conditional branch is then "branch if zero," (BZ) or "branch if one" (BO). The byte that contains the tested bits is called FLAG, and is defined elsewhere in the program. The bits tested are named C1, C2, and so forth, which means I can use the flowchart without modifying it.

The unconditional branch is simply B.

"Branch and link" is equivalent to a GOSUB in BASIC. A, B, etc., are the names of subroutines to which the program transfers control. They're also built elsewhere. Trace the flowchart with your finger as you read the code. It isn't challenging.

Notice that you sometimes have to define intermediate branch locations when you're coding something this complicated. That happens in BASIC, too, if you code explicitly. It doesn't cause a problem, but you'll rarely if ever want to code anything this complicated in any language.

Stylistically, I always code my branch locations as IF001, LOOP002, and so forth. I do believe in heavily commenting data locations, but for branch locations, this is about as meaningful as it ever gets. It is considerably more meaningful in terms of what the program is doing than the run of branch location naming methods.

I find the level of difficulty for this assembler and BASIC to be about the same. Neither is hard to code in, nor is it difficult to analyze existing assembler programs using the methods of Chapters III and IV. Those methods, in fact, first evolved as a means of making assembler programming easier. They do.

COBOL

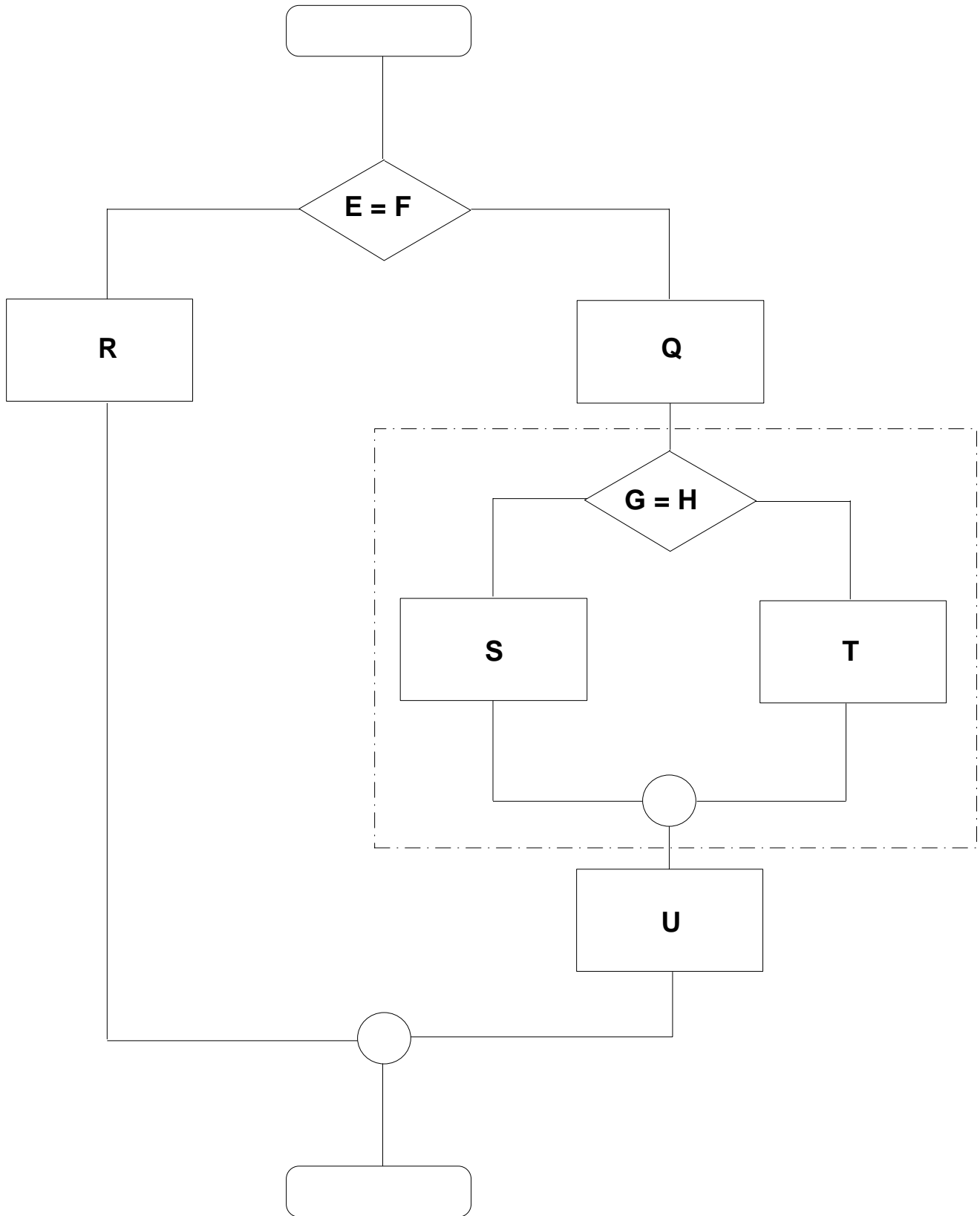
COBOL is generally harder to code in than assembler language. For one thing, COBOL doesn't allow you to put comments on the same line as the instruction, which makes it difficult to psuedocode. This is because COBOL was the first of the "self-documenting" languages, the general characteristics of which we covered in Chapter VI.

Another problem you run into with COBOLs that were standard before 1987 or so is illustrated in Figure B-2. COBOL has an IF-THEN-ENDIF structure, but the ENDIF is a period, intended to keep the language as "English-like" as possible.

The problem is that you can nest COBOL IF statements inside other COBOL IF statements, but when you write the period, it ends the entire nesting. This is very awkward if you're trying to code a structure like the one in the Figure. You can't just end the IF inside the dashed box.

Programmers responded to this problem in two ways: One was to put the code corresponding to the dashed box IF in a separate subroutine. This created thousands of programs with two- and three-line subroutines that read like a manual written in footnotes. It was primarily this appalling situation I alluded to when I warned against writing programs this way. COBOL, through its syntax, encourages this practice, though, and you will encounter its sad result in the field.

The other way programmers dealt with this quirk was to ignore the IF-THEN-ELSE-PERIOD instructions altogether, achieving the same effect with GO TOs as you do in BASIC. They programs



Programming

they got doing that are actually easier to maintain and modify than their more highly “structured” counterparts. Keeping track of the period is still a dicey business, but you can get the job done.

You have to be careful, then, not to get lost in detail to the extent you forget what your program is all about. Still, most of the methods for coding BASIC or assembler language programs work for COBOL as well, and COBOL programs are as amenable as any other to analysis and maintenance. Eventually, you just get used to it.