Software Tools in Pascal

Brian W. Kernighan

P. J. Plauger

Software Tools in Pascal

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey

P. J. Plauger

Whitesmiths, Ltd.
New York, New York

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts
Menlo Park, California · London · Amsterdam · Don Mills, Ontario · Sydney

This book was set in Times Roman and Courier by the authors, using a Mergenthaler Linotron 202 phototypesetter driven by a PDP-11/70 running the Unix operating system.

ISBN 0-201-10342-7
ABCDEFGHIJ-DO-8987654321

PREFACE

This book teaches how to write good programs that make good tools, by presenting a comprehensive set, each of which provides lessons in design and implementation. The programs are not artificial, nor are they toys. Instead, they are tools that have proved valuable in the production of other programs. We use most of them every working day, and they account for much of our computer usage. The programs are complete, not just algorithms and outlines, and they work: all have been tested directly from the text, which is in machine-readable form. They are readable: all are presented in standard Pascal. They are documented, so they can be used. Most important, the programs are designed to work well with people and with each other, and are thus *perceived* as tools.

The book is pragmatic. We teach top-down design by walking through designs. We demonstrate structured programming with structured programs. We discuss efficiency and reliability in terms of actual tests carried out. We illustrate documentation by presenting it for each program. We treat portability by writing in a language that is widely available, and by isolating unavoidable system dependencies in a handful of small, carefully specified routines that can be readily built for a particular operating environment. All of the programs presented here have been run without change on at least three different machines and several different Pascal compilers and interpreters. The code is available in machine-readable form as a supplement to the text.

The principles of good programming are presented not as abstract sermons but as concrete lessons in the context of actual working programs. For example, there is no chapter on "efficiency." Instead, throughout the book there are observations on efficiency as it relates to the particular program being developed. Similarly there is no chapter on "top-down design," nor on "structured programming," nor on "testing and debugging." Instead, all of these disciplines are employed as appropriate in every program shown.

The book is suitable for a "software engineering" course or for a second course in programming — more so, we feel, than the traditional dose of "compilers, assemblers and loaders," for the programs presented here are more of the size and nature that will be encountered by most programmers. It is also suitable as a supplementary text in any programming course; the only prerequisite is programming experience in a high-level language. Professional programmers will find it a guide to good programming techniques and a source of proven, useful programs. Numerous exercises are provided to test comprehension and to extend the concepts and the programs presented in the text.

Software Tools was originally published in 1976 with the programs written in Ratfor, a language based on Fortran. Ratfor was implemented as a preprocessor; it provided Fortran with modern control flow statements like `if-else` and `while`, and some cosmetic improvements like symbolic constants and free-form input. The approach and the tools have proved sufficiently useful that many copies of them have been distributed, and there is a large, active user group.

Pascal is now the dominant teaching language for computer science courses, and is becoming widely used outside of universities as well. We feel that the lessons about the design and implementation of tools from the original book carry over intact to Pascal. Thus *Software Tools in Pascal* has a great deal of overlap with the Ratfor original. The same programs are present, except that there is no preprocessor chapter, since Pascal provides most of the sensible control flow and cosmetic improvements that Ratfor adds to Fortran. On those systems where Pascal needs augmentation, tools such as the macro and file inclusion processors serve as language preprocessors.

The programs here are not just transliterations into Pascal, however. Almost every program has been improved in some way. Pascal lets us do some things much better than is possible in Fortran. Recursion in particular is a boon. Quicksort and regular expression closure are much simpler when done recursively instead of with a stack or linked list; expression evaluation has been added to the macro processor.

Pascal data types are generally more suitable for the clear expression of algorithms. Records let us deal with a group of related variables as a unit. Subranges and enumerated types make it easier to constrain the set of legal values for variables, so that errors are detected sooner and the code is easier to read. And eight-character variable names are a lot less contorted than six.

Regrettably, though, standard Pascal is far from an ideal language; in many ways it is less suitable for writing large programs than Fortran is. Since there is no standard way to specify separate compilation, the growth of libraries to extend the language is stunted. Since the size of an array is part of its type in standard Pascal, it is hard to write general-purpose routines that process arrays of different sizes. The lack of own variables and initialization forces variables to have global scope where Fortran would make them local to a single routine. Finally, the operating system interface provided by Pascal is just as unsuitable as Fortran's, but the language makes it harder to escape to one's own.

There are versions of Pascal that deal with each of these problems, with some success, but each such extension is non-standard and rarely portable. Our code adheres to the standard; it will work everywhere. The price we pay is increased compilation time, sometimes involving the use of one or more preprocessing steps; larger load modules, to provide an extended environment in the absence of libraries; and slower execution time, because we have consistently traded efficiency for portability. Each of these areas is readily amenable to improvement, however, by tuning the system interface to each local environment.

Building on the work of others is the only way to make substantial progress in any field. Yet programmers reinvent programs for each new application instead of using what already exists. We hope that *Software Tools in Pascal* will instill a feeling for how to design and write good programs that can be widely used, how to use existing tools, and how to improve a given environment with maximum effect for minimum effort.

We are grateful to many friends for careful reading, perceptive criticism, and continuous cheerful support. Ron Hardin, John Linderman, Doug McIlroy, Rob Pike and Dennis Ritchie all spent many hours reading the manuscript and exercising the programs, and made invaluable suggestions on how to improve both. We deeply appreciate their efforts. Our thanks also to Al Aho, Doug Comer, Al Feuer, John Gannon, Peter Grogono, Dave Hanson, Debbie Scherrer, and Chris Van Wyk for helpful comments at various stages. Bill Joy and Andy Tanenbaum provided us with rock-solid Pascal compilers; Bill Joy also made it possible for us to time our programs. Chuck Howerton provided the impetus that got us started in the first place.

Finally, it is a pleasure to acknowledge our debt to the Unix operating system, developed at Bell Labs by Ken Thompson and Dennis Ritchie. We wrote the text, tested the programs, and typeset the manuscript, all within Unix. Many of the tools we describe are based on Unix models. Most important, the ideas and philosophy are based on our experience as Unix users. Of all the operating systems we have used, Unix is the only one that has been a positive help in getting a job done instead of an obstacle to be overcome. The worldwide acceptance of Unix indicates that we are not the only ones who feel this way.

Brian W. Kernighan

P. J. Plauger

CONTENTS

All of the programs described in this book are available in machine-readable form from Addison-Wesley.

INTRODUCTION

We are going to discuss two things in this book — how to write programs that make good tools, and how to program well in the process.

What do we mean by a *tool*? Suppose you have a 5000-line Pascal program and you need to find all references to the variable `time`, to make sure it can safely be changed from type `integer` to type `real`. How would you do it?

One possibility is to get a listing and mark it up with a red pencil. But it doesn't take much imagination to see what's wrong with red-penciling a hundred pages of computer paper. It's mindless and boring busy-work, with lots of opportunities for error. And even after you've found all instances of `time`, you still can't do much, because the red marks aren't machine readable.

Another approach is to write a simple program to find lines containing the identifier `time`. This is an improvement, for such a program is faster and more accurate than doing the job by hand. The trouble is that the program is so specialized that it will be used once by its author, then tucked away and forgotten. No one else will benefit from the effort that went into writing it, and something very much like it will have to be reinvented for each new application.

Finding `time`'s in a Pascal program is a special case of a general problem, finding patterns in text. Whoever wanted references to `time` today will want references to some other variable tomorrow, `readln` and `writeln` calls the day after, and next week an entirely different pattern in some unrelated text. Red penciling never ends. The way to cope with the general problem is to provide a general purpose pattern finder that will look for a specified pattern and print all the lines where it occurs. Then anyone can say

```
find pattern
```

and the job is done. `find` is a *tool:* it uses the machine; it solves a general problem, not a special case; and it's so easy to use that people will use it instead of building their own.

Far too many programmers are red pencillers. Some are literal red pencillers who do things by hand that should be done by machine. Others are figurative red pencillers whose use of the machine is so clumsy and awkward that it might as well be manual. One purpose of this book is to show how to build *tools* —

programs to help people to do things by machine instead of by red pencil, and how to do them well instead of badly. We're going to do this, not by talking in generalities but by writing real, working programs, programs that we know from experience are useful tools. *Every* program in this book has been run and carefully tested, directly from the text itself, which is in machine-readable form. All of them have been run without change on a variety of machines and Pascal compilers.

The second concern of this book is how to write *good* programs. As we proceed, we hope to convey to you principles of: good design, so you write programs that work and are easy to maintain and modify; human engineering, so you can use them conveniently; reliability, so you get the right answers; and efficiency, so you can afford to run them.

We don't think that it is possible to learn to program well by reading platitudes about good programming. Nor is it sufficient to study small examples. Rather than present ideas like structured programming and top-down design as abstract principles, we have tried to distill the important contributions of each and put them into practice in all our code. That way you can see what they mean, how to use them on real problems, and what benefits they are likely to produce.

We also try to show *how* we went about building the programs, rather than just presenting the finished product, or pretending that we arrived at the final result by some mechanical process. For each program we discuss its purpose, how it should be designed to be easy to use, what considerations affect its structure and implementation, and some of the alternatives that exist. We don't claim that these are the best possible programs, or that our way is the only way to design and write them. But even if you would do them differently, studying the development of a coherent set of well-written and useful programs should help you better appreciate the significance of some of these ideas, and ultimately to become a better programmer.

We have quite a few tools to show you. Most of these are programs of manageable size, programs that one person can reasonably write in an hour or a day or a week. Clearly we can't present giant programs like operating systems or major compilers; few of us have the time, training or need to delve inside such creatures anyway. Instead we have concentrated on the kinds of tools you *are* likely to become involved with, programs that help you to make the most effective use of whatever operating system and language you already have. There is an important lesson in this: well chosen and well designed programs of modest size can be used to create a comfortable and effective interface to those that are bigger and less well done.

Whenever possible we will build more complicated programs up from the simpler; whenever possible we will *avoid* building at all, by finding new uses for existing tools, singly or in combinations. Our programs *work together:* their cumulative effect is much greater than you could get from a similar collection of programs that you couldn't easily connect. By the end of the book you will have been introduced to a set of tools that solve many problems you encounter as a programmer.

What sorts of tools? Computing is a broad field, and we can't begin to cover every kind of application. Instead we have concentrated on an activity that is central to programming — *programs that help develop other programs*. They are programs which we use regularly, most of them every day; we used versions of almost all of them while we were writing this book. In fact we chose them because they account for much of the computer usage on the system where we work. Although we can hardly claim that our choices will satisfy all your needs, some should be directly useful to you whatever your interest. Studying those that are not should provide you with ideas and insights about how to design and build quality tools for *your* particular problems. Comparing our designs with related programs on your system may lead you to improvements in both. And learning to think in terms of tools will encourage you to write programs that solve only the unique parts of your problem, then interface to existing programs to do the rest.

Whatever your application, your most important tool is a good programming language. Without this, programs are just too hard to write and understand; you spend more time fighting your language than being productive. One of the problems with writing about programming is choosing a language for the programs. No single language is known to all readers, available on all machines, and easy to read. We must compromise.

Since Pascal is widely available and well supported, we will use it as our base in this book. Pascal is now the main language in university computer science courses. It is available on almost all computers, and is sufficiently standardized that programs can be written to run without change on a wide variety of systems.

Most programmers can quickly achieve at least a reading knowledge of Pascal. If you are used to some other language, you should have no difficulty following our programs, for properly structured programs seem to read the same in most languages. We avoid most idiosyncrasies of Pascal, and hide the unavoidable ones in well-defined modules.

Although we are not writing a Pascal manual, we will try to explain new constructions as they arise. Chapter 1 describes a few simple tools, as a way to introduce our style of Pascal code and our conventions.

A surprising number of programs have one input, one output, and perform a useful transformation on data as it passes through. We call such programs *filters*. Some filters are so simple that you might hardly think of them as tools, yet a careful selection of filters that work together can handle quite complicated processing. Several smaller filters are collected in Chapter 2, including a powerful character transliteration program.

Not all programs are filters. Chapter 3 discusses programs that interact with their environment in more complicated ways, such as file inclusion, comparison and printing, and an archive system for managing sets of files. The major problem in moving programs from one environment to another is precisely this question of how a program communicates with its local operating system. We deal with portability by specifying a small set of primitive operations for accessing the environment. All of our programs are written in terms of these primitives, so operating system dependencies are confined to a handful of procedures and functions. Programs that use them can move to any system where the primitives can be implemented. We have demonstrated this by moving all of the tools in this book, without change, to several distinct Pascal systems on three different computers.

Some filters are large enough to warrant separate chapters. The sorting program of Chapter 4, the pattern finding and replacement programs of Chapter 5, and the macro processor of Chapter 8 all fall into this category. The pattern finder uses most of the code of the transliteration program in Chapter 2 to recognize character classes, which are just one of a larger set of patterns that can be specified. (The pattern finder is capable of a lot more than finding instances of `time`, by the way.) Although these filters are biased toward program development, the filter concept is valuable in any application. It encourages the view that a program is just a stage in a larger process, and that stages should be simple and easy to connect. It also encourages the view that all files and I/O devices should be interchangeable, so that any program can work with any file or device.

Chapter 6 contains a text editor that is rather more comprehensive than those normally found in time-sharing systems. The editor incorporates most of the code of the pattern finder of Chapter 5, so it recognizes the same class of patterns. When used with some of the other programs presented, it can do jobs that would otherwise require you to write a special program. Even if you are not working in an interactive environment, the editor will prove to be useful.

Chapter 7 contains a text formatter that is a (much smaller) version of the program used to set the type for this book.

Finally, as we have already mentioned, Chapter 8 contains a modest but useful macro processor, which you can use to extend any programming language.

It might appear from this outline that we stress text manipulation too heavily. Yet a large part of what programmers do every day *is* text processing — editing program source, preparing input data, scanning output, writing documentation. These activities are at the heart of programming; as much as possible, they should be mechanized. Program development is the place where tools can have the most impact. And since text processing programs come in all sizes, they display at least as broad a spectrum of programming techniques as language processors or numerical programs.

As you can see, the book is organized in terms of applications rather than different aspects of the programming process. This is not a reference work on algorithms or data structures or Pascal. Nor will you find separate chapters on design, coding, testing, debugging, efficiency, human engineering, documentation, or any of the other popular themes. We are engaged in the business of building tools, and of building them properly. All of these aspects of programming arise, in varying degrees, with *every* program, and can be kept in perspective only by discussing them as we write the programs. In the process, we will try to communicate to you our approach to tool building, so you can go on to design, build, and use tools of your own.

Bibliographic Notes

The programming language Pascal has had considerable impact on computing practice; it is especially suitable for structured programming and for describing data structures. Read *Systematic Programming: An Introduction* (Prentice-Hall, 1973) by N. Wirth, the designer of Pascal. The special issue of *Computing Surveys* on programming (December, 1974) contains several papers well worth reading, including one by Wirth. Pascal has also influenced the design of newer languages, most notably Ada; you might read "An Overview of Ada," by J. G. P. Barnes, *Software Practice and Experience*, November, 1980, or *Programming with Ada* by P. Wegner (Prentice-Hall, 1980).

The Pascal language is defined in *Pascal User Manual and Report (2nd Edition)*, by K. Jensen and N. Wirth (Springer-Verlag, 1978), and with more detail and precision in the proposed ISO Standard for Pascal. (See, for example, SIGPLAN Notices, April, 1980.)

One of the most influential proponents of good programming is E. W. Dijkstra. You should read *Structured Programming*, by O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare (Academic Press, 1972) and Dijkstra's *A Discipline of Programming* (Prentice-Hall, 1976).

An excellent set of essays on programming and on the problems of developing big systems is found in F. P. Brooks' *The Mythical Man-Month* (Addison-Wesley, 1974). The term "egoless programming" was coined by G. M. Weinberg in his delightful book *The Psychology of Computer Programming* (Van Nostrand Reinhold, 1971).

The Elements of Programming Style (2nd Edition), by B. W. Kernighan and P. J. Plauger (McGraw-Hill, 1978), contains an extensive discussion of how to improve computer programs, with numerous examples taken from published Fortran and PL/I code.

The original version of this book is *Software Tools* (Addison-Wesley, 1976). The programs therein are written in Ratfor, a structured dialect of Fortran implemented by a preprocessor. They have proved sufficiently popular that a user group exists, and the tools themselves are used at several thousand sites. See "A Virtual Operating System," by D. E. Hall, D. K. Scherrer and J. S. Sventek, *CACM*, September, 1980. With the major exception of the Ratfor preprocessor itself, all of the tools from the original are presented here.

[illegible] We have demonstrated this by running all of the tools in this book, without change, on several different Pascal compilers [illegible]

[illegible] Pascal User Manual and Report [illegible] K. Jensen and N. Wirth (Springer-Verlag [illegible]) [illegible] proposed ISO standard for Pascal [illegible]

One of the most influential discussions of good programming is [illegible] Structured Programming, by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Academic Press, 1972) [illegible]

[illegible] The term "egoless programming" was coined by G. M. Weinberg in his delightful book, The Psychology of Computer Programming (Van Nostrand Reinhold, 1971).

The Elements of Programming Style, by B. W. Kernighan and P. J. Plauger (McGraw-Hill, 1978) [illegible]

The original version of this book is [illegible]

[illegible] "A Virtual Operating System" [illegible]

CHAPTER 1: GETTING STARTED

This chapter is an informal introduction to our way of using Pascal, and to some of the ideas and conventions used throughout the book. It also presents a handful of small but useful programs, to make the discussion concrete. We cannot present complete programs without occasionally using concepts before they are explained, so you will have to take some things on faith as we get started or we'll get bogged down explaining our explanations. Bear with us.

1.1 File Copying

The first problem we want to tackle is how a program communicates with its environment. Since many of our programs are concerned with text manipulation, one basic operation is reading characters from some source of input. To do this we will invent a function called `getc`, which reads the *next input character*, and returns that character as its function value; each time it is called, it returns a new character. For now we'll ignore where the characters come from, although you can imagine them originating at an interactive terminal or some secondary storage device like a disk.

We won't discuss what character set we have in mind, except to say that `getc` can return a value, distinguishable from all input character codes, that indicates that the end of the input has been reached. Similarly, the end of a text line is indicated by yet another unique value that is returned by `getc`. We'll also ignore all questions of efficiency, although we're fully aware that reading one character at a time at least sounds expensive. Temporarily we want to sweep as many details as we possibly can under the rug.

Next we invent `putc`, the complement of `getc`. `putc` puts a single character somewhere, such as a terminal, a printer, or a disk; one of its acceptable argument values signals the end of a text line. Again, we won't concern ourselves with the precise details, nor with the efficiency of the operation. The main point is that `getc` and `putc` work together — the characters that `getc` gets can be put somewhere else by `putc`.

If someone has provided these two basic operations, you can do a surprising amount of useful computing without ever knowing anything more about how they're implemented. As the simplest example, if you put the `getc/putc` pair inside a loop:

```
while (getc(c) is not at end of input) do
     putc(c)
```

you have a program that copies its input to its output and quits. A simple task, performed by an equally simple program. Certainly, someone ultimately has to worry about the choice of character set, detecting end of line and end of input, efficiency and the like, but most people need *not* be concerned, because `getc` and `putc` conceal the details. (If you want to know how they might work, we will show you simple versions in standard Pascal soon, and also explain why we didn't just use `read` and `write`.)

Functions like `getc` and `putc` are called *primitives* — functions that interface to the "outside world." They call in turn whatever input and output routines must be used with a particular operating system and compiler. To the program that uses them, `getc` and `putc` define a standard internal representation for characters and provide an input-output mechanism that can be made uniform across many different computers. If we use primitives, we can design and write programs that will not be overly dependent on the idiosyncrasies of any one operating system. The primitives insulate a program from its operating system environment and ensure that the high level task to be performed is clearly expressed in a small well-defined set of basic operations.

The program shown above is written in "pseudo-code," that is, a language that resembles a real programming language but avoids excessive detail by from time to time resorting to ordinary English. Writing in pseudo-code lets us specify quite a bit of the program before we have worked out all aspects of it. On larger programs, it is valuable to begin with pseudo-code and refine it in stages until it is all executable. You can revise and improve the design at a high level without writing any executable code, yet remain close to a form that can be made executable.

The next step is to write `copy` in standard Pascal, ready to compile and run.

```
{ copy -- copy input to output }
procedure copy;
var
    c : character;
begin
    while (getc(c) <> ENDFILE) do
        putc(c)
end;
```

Some explanations: First, and most obvious to people who have used Pascal before, is that this is not a complete program — it is just a procedure. So it needs some surrounding context before it can actually do anything for us. We intend to present all of our programs this way, as procedures that fit into a larger context, so we can better focus on the essential ideas. To make `copy` run, we actually need something like this:

```
{ complete copy -- to show one possible implementation }
program copyprog (input, output);
const
    ENDFILE = -1;
    NEWLINE = 10;   { ASCII value }
type
    character = -1..127;    { ASCII, plus ENDFILE }

{ getc -- get one character from standard input }
function getc (var c : character) : character;
var
    ch : char;
begin
    if (eof) then
        c := ENDFILE
    else if (eoln) then begin
        readln;
        c := NEWLINE
    end
    else begin
        read(ch);
        c := ord(ch)
    end;
    getc := c
end;

{ putc -- put one character on standard output }
procedure putc (c : character);
begin
    if (c = NEWLINE) then
        writeln
    else
        write(chr(c))
end;

{ copy -- copy input to output }
procedure copy;
var
    c : character;
begin
    while (getc(c) <> ENDFILE) do
        putc(c)
end;

begin  { main program }
    copy
end.
```

The context shown here defines all the constants, types, and functions needed by `copy`. It is presented in standard Pascal to illustrate the behavior of `getc` and putc in terms familiar to Pascal programmers, and to demonstrate that the primitives can be implemented in a fashion that is supported on all Pascal systems. For most implementations, however, some special treatment would be given to getc and putc, to make them as efficient as possible.

The advantage of wrapping a program in an outer shell is that we can gradually add to the surrounding environment as we make the programs more sophisticated, without having to repeat a lot of description every time we present a new program. The standard context for the programs in the book is much larger than what we showed here. In particular, we put the definitions of functions and procedures like getc and putc, constants like ENDFILE, and types like character in the outer block so they are readily available to the whole program. In Chapter 3 and Chapter 8, we will show some programs that help to automate collecting the pieces of a program. The appendix shows the declarations we use. We will assume without further comment that all subsequent programs are wrapped up this way.

Now back to copy itself. The first line is just a comment, of course, that says briefly what the procedure does. This kind of comment will occur on every function and procedure in the book. We use { and } to delimit comments; you may have to use (* and *) if your character set does not include braces.

The lines

```
var
    c : character;
```

declare c to be a variable of type character. Note that character is not the same as the standard type char, for it must represent values like ENDFILE that must be different from legitimate values of type char.

The lines

```
    while (getc(c) <> ENDFILE) do
        putc(c)
```

are where all the work of copy gets done. The while statement specifies a loop; so long as the condition inside parentheses† is true, the body of the loop (in this case, the single statement putc(c)) is repeatedly executed. Eventually the condition becomes false, and the loop terminates. copy then returns to its caller, and the whole program terminates. The condition being tested in the while loop is

```
getc(c) <> ENDFILE
```

The notation <> means "not equal to," so the loop continues while the character returned by getc is not ENDFILE.

† Strictly speaking, parentheses aren't needed here, but they are in conditions that involve and and or. We intend to stick them in everywhere because it's easier than remembering when they're needed and when they're not.