AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 100

Why Pascal is Not My Favorite Programming Language

Brian W. Kernighan

April 2, 1981

Why Pascal is Not My Favorite Programming Language

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The programming language Pascal has become the dominant language of instruction in computer science education. It has also strongly influenced languages developed subsequently, in particular Ada.

Pascal was originally intended primarily as a teaching language, but it has been more and more often recommended as a language for serious programming as well, for example, for system programming tasks and even operating systems.

Pascal, at least in its standard form, is just plain not suitable for serious programming. This paper discusses my personal discovery of some of the reasons why.

April 2, 1981

Why Pascal is Not My Favorite Programming Language

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Genesis

This paper has its origins in two events — a spate of papers that compare C and Pascal[1, 2, 3, 4] and a personal attempt to rewrite *Software Tools*[5] in Pascal.

Comparing C and Pascal is rather like comparing a Learjet to a Piper Cub — one is meant for getting something done while the other is meant for learning — so such comparisons tend to be somewhat farfetched. But the revision of *Software Tools* seems a more relevant comparison. The programs therein were originally written in Ratfor, a ''structured'' dialect of Fortran implemented by a preprocessor. Since Ratfor is really Fortran in disguise, it has few of the assets that Pascal brings — data types more suited to character processing, data structuring capabilities for better defining the organization of one's data, and strong typing to enforce telling the truth about the data.

It turned out to be harder than I had expected to rewrite the programs in Pascal. This paper is an attempt to distill out of the experience some lessons about Pascal's suitability for programming (as distinguished from learning about programming). It is *not* a comparison of Pascal with C or Ratfor.

The programs were first written in that dialect of Pascal supported by the Pascal interpreter *pi* provided by the University of California at Berkeley. The language is close to the nominal standard of Jensen and Wirth,[6] with good diagnostics and careful run-time checking. Since then, the programs have also been run, unchanged except for new libraries of primitives, on four other systems: an interpreter from the Free University of Amsterdam (hereinafter referred to as VU, for Vrije Universiteit), a VAX version of the Berkeley system (a true compiler), a compiler purveyed by Whitesmiths, Ltd., and UCSD Pascal on a Z80. All but the last of these Pascal systems are written in C.

Pascal is a much-discussed language. A recent bibliography[7] lists 175 items under the heading of ''discussion, analysis and debate.'' The most often cited papers (well worth reading) are a strong critique by Habermann[8] and an equally strong rejoinder by Lecarme and Desjardins.[9] The paper by Boom and DeJong[10] is also good reading. Wirth's own assessment of Pascal is found in [11]. I have no desire or ability to summarize the literature; this paper represents my personal observations and most of it necessarily duplicates points made by others. I have tried to organize the rest of the material around the issues of

types and scope
control flow
environment
cosmetics

and within each area more or less in decreasing order of significance.

To state my conclusions at the outset: Pascal may be an admirable language for teaching beginners how to program; I have no first-hand experience with that. It was a considerable achievement for 1968. It has certainly influenced the design of recent languages, of which Ada is likely to be the most important. But in its standard form (both current and proposed), Pascal is not adequate for writing real programs. It is suitable only for small, self-contained programs that have only trivial interactions with their environment and that make no use of any software written by anyone else.

2. Types and Scopes

Pascal is (almost) a strongly typed language. Roughly speaking, that means that each object in a program has a well-defined type which implicitly defines the legal values of and operations on the object. The language guarantees that it will prohibit illegal values and operations, by some mixture of compile- and run-time checking. Of course compilers may not actually do all the checking implied in the language definition. Furthermore, strong typing is *not* to be confused with dimensional analysis. If one defines types `apple` and `orange` with

```
type
        apple = integer;
        orange = integer;
```

then any arbitrary arithmetic expression involving `apples` and `oranges` is perfectly legal.

Strong typing shows up in a variety of ways. For instance, arguments to functions and procedures are checked for proper type matching. Gone is the Fortran freedom to pass a floating point number into a subroutine that expects an integer; this I deem a desirable attribute of Pascal, since it warns of a construction that will certainly cause an error.

Integer variables may be declared to have an associated range of legal values, and the compiler and run-time support ensure that one does not put large integers into variables that only hold small ones. This too seems like a service, although of course run-time checking does exact a penalty.

Let us move on to some problems of type and scope.

2.1. The size of an array is part of its type

If one declares

```
var     arr10 : array [1..10] of integer;
        arr20 : array [1..20] of integer;
```

then `arr10` and `arr20` are arrays of 10 and 20 integers respectively. Suppose we want to write a procedure `sort` to sort an integer array. Because `arr10` and `arr20` have different types, it is not possible to write a single procedure that will sort them both.

The place where this affects *Software Tools* particularly, and I think programs in general, is that it makes it difficult indeed to create a library of routines for doing common, general-purpose operations like sorting.

The particular data type most often affected is `array of char`, for in Pascal a string is an array of characters. Consider writing a function `index(s,c)` that will return the position in the string `s` where the character `c` first occurs, or zero if it does not. The problem is how to handle the string argument of `index`. The calls `index('hello',c)` and `index('goodbye',c)` cannot both be legal, since the strings have different lengths. (I pass over the question of how the end of a constant string like `'hello'` can be detected, because it can't.)

The next try is

```
var temp : array [1..10] of char;

temp := 'hello';

n := index(temp,c);
```

but the assignment to `temp` is illegal because `'hello'` and `temp` are of different lengths.

The only escape from this infinite regress is to define a family of routines with a member for each possible string size, or to make all strings (including constant strings like `'define'`) of the same length.

The latter approach is the lesser of two great evils. In *Tools*, a type called `string` is declared as

```
type string = array [1..MAXSTR] of char;
```

where the constant `MAXSTR` is ''big enough,'' and all strings in all programs are exactly this size. This is far from ideal, although it made it possible to get the programs running. It does *not* solve the problem of creating true libraries of useful routines.

There are some situations where it is simply not acceptable to use the fixed-size array representation. For example, the *Tools* program to sort lines of text operates by filling up memory with as many lines as will fit; its running time depends strongly on how full the memory can be packed. Thus for `sort`, another representation is used, a long array of characters and a set of indices into this array:

```
type    charbuf = array [1..MAXBUF] of char;
        charindex = array [1..MAXINDEX] of 0..MAXBUF;
```

But the procedures and functions written to process the fixed-length representation cannot be used with the variable-length form; an entirely new set of routines is needed to copy and compare strings in this representation. In Fortran or C the same functions could be used for both.

As suggested above, a constant string is written as

```
'this is a string'
```

and has the type `packed array [1..n] of char`, where `n` is the length. Thus each string literal of different length has a different type. The only way to write a routine that will print a message and clean up is to pad all messages out to the same maximum length:

```
error('short message                    ');
error('this is a somewhat longer message');
```

Many commercial Pascal compilers provide a `string` data type that explicitly avoids the problem; `string`'s are all taken to be the same type regardless of size. This solves the problem for this single data type, but no other. It also fails to solve secondary problems like computing the length of a constant string; another built-in function is the usual solution.

Pascal enthusiasts often claim that to cope with the array-size problem one merely has to copy some library routine and fill in the parameters for the program at hand, but the defense sounds weak at best:[12]

> ''Since the bounds of an array are part of its type (or, more exactly, of the type of its indexes), it is impossible to define a procedure or function which applies to arrays with differing bounds. Although this restriction may appear to be a severe one, the experiences we have had with Pascal tend to show that it tends to occur very infrequently. [...] However, the need to bind the size of parametric arrays is a serious defect in connection with the use of program libraries.''

This botch is the biggest single problem with Pascal. I believe that if it could be fixed, the language would be an order of magnitude more usable. The proposed ISO standard for Pascal[13] provides such a fix (''conformant array schemas''), but the acceptance of this part of the standard is apparently still in doubt.

2.2. There are no static variables and no initialization

A *static* variable (often called an *own* variable in Algol-speaking countries) is one that is private to some routine and retains its value from one call of the routine to the next. *De facto,* Fortran variables are internal static, except for COMMON;† in C there is a `static` declaration that can be applied to local variables.

---

† Strictly speaking, in Fortran 77 one must use SAVE to force the static attribute.

Pascal has no such storage class. This means that if a Pascal function or procedure intends to remember a value from one call to another, the variable used must be external to the function or procedure. Thus it must be visible to other procedures, and its name must be unique in the larger scope. A simple example of the problem is a random number generator: the value used to compute the current output must be saved to compute the next one, so it must be stored in a variable whose lifetime includes all calls of the random number generator. In practice, this is typically the outermost block of the program. Thus the declaration of such a variable is far removed from the place where it is actually used.

One example comes from the text formatter described in Chapter 7 of *Tools.* The variable `dir` controls the direction from which excess blanks are inserted during line justification, to obtain left and right alternately. In Pascal, the code looks like this:

```
program formatter (...);

var
        dir : 0..1;     { direction to add extra spaces }
        .
        .
        .
procedure justify (...);
begin
        dir := 1 - dir; { opposite direction from last time }
        ...
end;

        ...

begin { main routine of formatter }
        dir := 0;
        ...
end;
```

The declaration, initialization and use of the variable `dir` are scattered all over the program, literally hundreds of lines apart. In C or Fortran, `dir` can be made private to the only routine that needs to know about it:

```
        ...
main()
{
        ...
}

        ...

justify()
{
        static int dir = 0;

        dir = 1 - dir;
        ...
}
```

There are of course many other examples of the same problem on a larger scale; functions for buffered I/O, storage management, and symbol tables all spring to mind.

There are at least two related problems. Pascal provides no way to initialize variables statically (i.e., at compile time); there is nothing analogous to Fortran's `DATA` statement or initializers like

```
int dir = 0;
```

in C. This means that a Pascal program must contain explicit assignment statements to initialize variables (like the

```
dir := 0;
```

above). This code makes the program source text bigger, and the program itself bigger at run time.

Furthermore, the lack of initializers exacerbates the problem of too-large scope caused by the lack of a static storage class. The time to initialize things is at the beginning, so either the main routine itself begins with a lot of initialization code, or it calls one or more routines to do the initializations. In either case, variables to be initialized must be visible, which means in effect at the highest level of the hierarchy. The result is that any variable that is to be initialized has global scope.

The third difficulty is that there is no way for two routines to share a variable unless it is declared at or above their least common ancestor. Fortran COMMON and C's external static storage class both provide a way for two routines to cooperate privately, without sharing information with their ancestors.

The new standard does not offer static variables, initialization or non-hierarchical communication.

2.3. Related program components must be kept separate

Since the original Pascal was implemented with a one-pass compiler, the language believes strongly in declaration before use. In particular, procedures and functions must be declared (body and all) before they are used. The result is that a typical Pascal program reads from the bottom up — all the procedures and functions are displayed before any of the code that calls them, at all levels. This is essentially opposite to the order in which the functions are designed and used.

To some extent this can be mitigated by a mechanism like the `#include` facility of C and Ratfor: source files can be included where needed without cluttering up the program. `#include` is not part of standard Pascal, although the UCB, VU and Whitesmiths compilers all provide it.

There is also a `forward` declaration in Pascal that permits separating the declaration of the function or procedure header from the body; it is intended for defining mutually recursive procedures. When the body is declared later on, the header on that declaration may contain only the function name, and must not repeat the information from the first instance.

A related problem is that Pascal has a strict order in which it is willing to accept declarations. Each procedure or function consists of

```
label     label declarations, if any
const     constant declarations, if any
type      type declarations, if any
var       variable declarations, if any
procedure and function declarations, if any
begin
          body of function or procedure
end
```

This means that all declarations of one kind (types, for instance) must be grouped together for the convenience of the compiler, even when the programmer would like to keep together things that are logically related so as to understand the program better. Since a program has to be presented to the compiler all at once, it is rarely possible to keep the declaration, initialization and use of types and variables close together. Even some of the most dedicated Pascal supporters agree:[14]

> "The inability to make such groupings in structuring large programs is one of Pascal's most frustrating limitations."

A file inclusion facility helps only a little here.

The new standard does not relax the requirements on the order of declarations.

2.4. There is no separate compilation

The ''official'' Pascal language does not provide separate compilation, and so each implementation decides on its own what to do. Some (the Berkeley interpreter, for instance) disallow it entirely; this is closest to the spirit of the language and matches the letter exactly. Many others provide a declaration that specifies that the body of a function is externally defined. In any case, all such mechanisms are non-standard, and thus done differently by different systems.

Theoretically, there is no need for separate compilation — if one's compiler is very fast (and if the source for all routines is always available and if one's compiler has a file inclusion facility so that multiple copies of source are not needed), recompiling everything is equivalent. In practice, of course, compilers are never fast enough and source is often hidden and file inclusion is not part of the language, so changes are time-consuming.

Some systems permit separate compilation but do not validate consistency of types across the boundary. This creates a giant hole in the strong typing. (Most other languages do no cross-compilation checking either, so Pascal is not inferior in this respect.) I have seen at least one paper (mercifully unpublished) that on page *n* castigates C for failing to check types across separate compilation boundaries while suggesting on page *n*+1 that the way to cope with Pascal is to compile procedures separately to avoid type checking.

The new standard does not offer separate compilation.

2.5. Some miscellaneous problems of type and scope

Most of the following points are minor irritations, but I have to stick them in somewhere.

It is not legal to name a non-basic type as the literal formal parameter of a procedure; the following is not allowed:

```
procedure add10 (var a : array [1..10] of integer);
```

Rather, one must invent a type name, make a type declaration, and declare the formal parameter to be an instance of that type:

```
type    a10 = array [1..10] of integer;
...
procedure add10 (var a : a10);
```

Naturally the type declaration is physically separated from the procedure that uses it. The discipline of inventing type names is helpful for types that are used often, but it is a distraction for things used only once.

It is nice to have the declaration `var` for formal parameters of functions and procedures; the procedure clearly states that it intends to modify the argument. But the calling program has no way to declare that a variable is to be modified — the information is only in one place, while two places would be better. (Half a loaf is better than none, though — Fortran tells the user nothing about who will do what to variables.)

It is also a minor bother that arrays are passed by value by default — the net effect is that every array parameter is declared `var` by the programmer more or less without thinking. If the `var` declaration is inadvertently omitted, the resulting bug is subtle.

Pascal's `set` construct seems like a good idea, providing notational convenience and some free type checking. For example, a set of tests like

```
if (c = blank) or (c = tab) or (c = newline) then ...
```

can be written rather more clearly and perhaps more efficiently as

```
if c in [blank, tab, newline] then ...
```

But in practice, set types are not useful for much more than this, because the size of a set is strongly implementation dependent (probably because it was so in the original CDC implementation: 59 bits). For example, it is natural to attempt to write the function `isalphanum(c)` (''is `c` alphanumeric?'') as

```
{ isalphanum(c) -- true if c is letter or digit }
function isalphanum (c : char) : boolean;
begin
        isalphanum := c in ['a'..'z', 'A'..'Z', '0'..'9']
end;
```

But in many implementations of Pascal (including the original) this code fails because sets are just too small. Accordingly, sets are generally best left unused if one intends to write portable programs. (This specific routine also runs an order of magnitude slower with sets than with a range test or array reference.)

2.6. There is no escape

There is no way to override the type mechanism when necessary, nothing analogous to the ''cast'' mechanism in C. This means that it is not possible to write programs like storage allocators or I/O systems in Pascal, because there is no way to talk about the type of object that they return, and no way to force such objects into an arbitrary type for another use. (Strictly speaking, there is a large hole in the type-checking near variant records, through which some otherwise illegal type mismatches can be obtained.)

3. Control Flow

The control flow deficiencies of Pascal are minor but numerous — the death of a thousand cuts, rather than a single blow to a vital spot.

There is no guaranteed order of evaluation of the logical operators `and` and `or` — nothing like && and `||` in C. This failing, which is shared with most other languages, hurts most often in loop control:

```
while (i <= XMAX) and (x[i] > 0) do ...
```

is extremely unwise Pascal usage, since there is no way to ensure that `i` is tested before `x[i]` is.

By the way, the parentheses in this code are mandatory — the language has only four levels of operator precedence, with relationals at the bottom.

There is no `break` statement for exiting loops. This is consistent with the one entry-one exit philosophy espoused by proponents of structured programming, but it does lead to nasty circumlocutions or duplicated code, particularly when coupled with the inability to control the order in which logical expressions are evaluated. Consider this common situation, expressed in C or Ratfor:

```
while (getnext(...)) {
        if (something)
                break
        rest of loop
}
```

With no `break` statement, the first attempt in Pascal is

```
done := false;
while (not done) and (getnext(...)) do
        if something then
                done := true
        else begin
                rest of loop
        end
```

But this doesn't work, because there is no way to force the ''`not done`'' to be evaluated before the next call of `getnext`. This leads, after several false starts, to

```
done := false;
while not done do begin
        done := getnext(...);
        if something then
                done := true
        else if not done then begin
                rest of loop
        end
end
```

Of course recidivists can use a `goto` and a label (numeric only and it has to be declared) to exit a loop. Otherwise, early exits are a pain, almost always requiring the invention of a boolean variable and a certain amount of cunning. Compare finding the last non-blank in an array in Ratfor:

```
for (i = max; i > 0; i = i - 1)
        if (arr(i) != ' ')
                break
```

with Pascal:

```
done := false;
i := max;
while (i > 0) and (not done) do
        if arr[i] = ' ' then
                i := i - 1
        else
                done := true;
```

The index of a `for` loop is undefined outside the loop, so it is not possible to figure out whether one went to the end or not. The increment of a `for` loop can only be `+1` or `-1`, a minor restriction.

There is no `return` statement, again for one in-one out reasons. A function value is returned by setting the value of a pseudo-variable (as in Fortran), then falling off the end of the function. This sometimes leads to contortions to make sure that all paths actually get to the end of the function with the proper value. There is also no standard way to terminate execution except by reaching the end of the outermost block, although many implementations provide a `halt` that causes immediate termination.

The `case` statement is better designed than in C, *except* that there is no `default` clause and the behavior is undefined if the input expression does not match any of the cases. This crucial omission renders the `case` construct almost worthless. In over 6000 lines of Pascal in *Software Tools in Pascal*, I used it only four times, although if there had been a `default`, a `case` would have served in at least a dozen places.

The new standard offers no relief on any of these points.

4. The Environment

The Pascal run-time environment is relatively sparse, and there is no extension mechanism except perhaps source-level libraries in the ''official'' language.

Pascal's built-in I/O has a deservedly bad reputation. It believes strongly in record-oriented input and output. It also has a look-ahead convention that is hard to implement properly in an interactive environment. Basically, the problem is that the I/O system believes that it must read one record ahead of the record that is being processed. In an interactive system, this means that when a program is started, its first operation is to try to read the terminal for the first line of input, before any of the program itself has been executed. But in the program

```
write('Please enter your name: ');
read(name);
...
```

read-ahead causes the program to hang, waiting for input before printing the prompt that asks for it.

It is possible to escape most of the evil effects of this I/O design by very careful implementation, but not all Pascal systems do so, and in any case it is relatively costly.

The I/O design reflects the original operating system upon which Pascal was designed; even Wirth acknowledges that bias, though not its defects.[15] It is assumed that text files consist of records, that is, lines of text. When the last character of a line is read, the built-in function `eoln` becomes true; at that point, one must call `readln` to initiate reading a new line and reset `eoln`. Similarly, when the last character of the file is read, the built-in `eof` becomes true. In both cases, `eoln` and `eof` must be tested before each `read` rather than after.

Given this, considerable pains must be taken to simulate sensible input. This implementation of `getc` works for Berkeley and VU I/O systems, but may not necessarily work for anything else:

```
{ getc -- read 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;
```

The type `character` is not the same as `char`, since `ENDFILE` and perhaps `NEWLINE` are not legal values for a `char` variable.

There is no notion at all of access to a file system except for predefined files named by (in effect) logical unit number in the `program` statement that begins each program. This apparently reflects the CDC batch system in which Pascal was originally developed. A file variable

```
var     fv : file of type
```

is a very special kind of object — it cannot be assigned to, nor used except by calls to built-in procedures like `eof`, `eoln`, `read`, `write`, `reset` and `rewrite`. (`reset` rewinds a file and makes it ready for re-reading; `rewrite` makes a file ready for writing.)

Most implementations of Pascal provide an escape hatch to allow access to files by name from the outside environment, but not conveniently and not standardly. For example, many systems permit a filename argument in calls to `reset` and `rewrite`:

```
reset(fv, filename);
```

But `reset` and `rewrite` are procedures, not functions — there is no status return and no way to regain control if for some reason the attempted access fails. (UCSD provides a compile-time flag that disables the normal abort.) And since `fv`'s cannot appear in expressions like

```
reset(fv, filename);
if fv = failure then ...
```

there is no escape in that direction either. This straitjacket makes it essentially impossible to write programs that recover from mis-spelled file names, etc. I never solved it adequately in the *Tools* revision.

There is no notion of access to command-line arguments, again probably reflecting Pascal's batch-processing origins. Local routines may allow it by adding non-standard procedures to the environment.

Since it is not possible to write a general-purpose storage allocator in Pascal (there being no way to talk about the types that such a function would return), the language has a built-in procedure called `new` that allocates space from a heap. Only defined types may be allocated, so it is not possible to allocate, for example, arrays of arbitrary size to hold character strings. The pointers returned by `new` may be passed around but not manipulated: there is no pointer arithmetic. There is no way to regain control if storage runs out.

The new standard offers no change in any of these areas.

5. Cosmetic Issues

Most of these issues are irksome to an experienced programmer, and some are probably a nuisance even to beginners. All can be lived with.

Pascal, in common with most other Algol-inspired languages, uses the semicolon as a statement separator rather than a terminator (as it is in PL/I and C). As a result one must have a reasonably sophisticated notion of what a statement is to put semicolons in properly. Perhaps more important, if one is serious about using them in the proper places, a fair amount of nuisance editing is needed. Consider the first cut at a program:

```
if a then
        b;
c;
```

But if something must be inserted before `b`, it no longer needs a semicolon, because it now precedes an `end`:

```
if a then begin
        b0;
        b
end;
c;
```

Now if we add an `else`, we *must* remove the semicolon on the `end`:

```
if a then begin
        b0;
        b
end
else
        d;
c;
```

And so on and so on, with semicolons rippling up and down the program as it evolves.

One generally accepted experimental result in programmer psychology is that semicolon as separator is about ten times more prone to error than semicolon as terminator.[16] (In Ada,[17] the most significant language based on Pascal, semicolon is a terminator.) Fortunately, in Pascal one can almost always close one's eyes and get away with a semicolon as a terminator. The exceptions are in places like declarations, where the separator vs. terminator problem doesn't seem as serious anyway, and just before `else`, which is easy to remember.

C and Ratfor programmers find `begin` and `end` bulky compared to `{` and `}`.

A function name by itself is a call of that function; there is no way to distinguish such a function call from a simple variable except by knowing the names of the functions. Pascal uses the Fortran trick of having the function name act like a variable within the function, except that where in Fortran the function name really is a variable, and can appear in expressions, in Pascal, its appearance in an expression is a recursive invocation: if `f` is a zero-argument function, `f:=f+1` is a recursive call of `f`.

There is a paucity of operators (probably related to the paucity of precedence levels). In particular, there are no bit-manipulation operators (AND, OR, XOR, etc.). I simply gave up trying to write the following trivial encryption program in Pascal:

```
i := 1;
while getc(c) <> ENDFILE do begin
        putc(xor(c, key[i]));
        i := i mod keylen + 1
end
```

because I couldn't write a sensible `xor` function. The set types help a bit here (so to speak), but not enough; people who claim that Pascal is a system programming language have generally overlooked this point. For example, [18, p. 685]

> ''Pascal is at the present time [1977] the best language in the public domain for purposes of system programming and software implementation.''

seems a bit naive.

There is no null string, perhaps because Pascal uses the doubled quote notation to indicate a quote embedded in a string:

```
'This is a '' character'
```

There is no way to put non-graphic symbols into strings. In fact, non-graphic characters are unpersons in a stronger sense, since they are not mentioned in any part of the standard language. Concepts like newlines, tabs, and so on are handled on each system in an *ad hoc* manner, usually by knowing something about the character set (e.g., ASCII newline has decimal value 10).

There is no macro processor. The `const` mechanism for defining manifest constants takes care of about 95 percent of the uses of simple `#define` statements in C, but more involved ones are hopeless. It is certainly possible to put a macro preprocessor on a Pascal compiler. This allowed me to simulate a sensible `error` procedure as

```
#define  error(s)  begin writeln(s); halt end
```

(`halt` in turn might be defined as a branch to the end of the outermost block.) Then calls like

```
        error('little string');
        error('much bigger string');
```

work since `writeln` (as part of the standard Pascal environment) can handle strings of any size. It is unfortunate that there is no way to make this convenience available to routines in general.

The language prohibits expressions in declarations, so it is not possible to write things like

```
const   SIZE = 10;
type            arr = array [1..SIZE+1] of integer;
```

or even simpler ones like

```
const   SIZE = 10;
                SIZE1 = SIZE + 1;
```

6. Perspective

The effort to rewrite the programs in *Software Tools* started in March, 1980, and, in fits and starts, lasted until January, 1981. The final product[19] was published in June, 1981. During that time I gradually adapted to most of the superficial problems with Pascal (cosmetics, the inadequacies of control flow), and developed imperfect solutions to the significant ones (array sizes, run-time environment).

The programs in the book are meant to be complete, well-engineered programs that do non-trivial tasks. But they do not have to be efficient, nor are their interactions with the operating system very complicated, so I was able to get by with some pretty kludgy solutions, ones that simply wouldn't work for real programs.

There is no significant way in which I found Pascal superior to C, but there are several places where it is a clear improvement over Ratfor. Most obvious by far is recursion: several programs are much cleaner when written recursively, notably the pattern-search, quicksort, and expression evaluation.

Enumeration data types are a good idea. They simultaneously delimit the range of legal values and document them. Records help to group related variables. I found relatively little use for pointers.

Boolean variables are nicer than integers for Boolean conditions; the original Ratfor programs contained some unnatural constructions because Fortran's logical variables are badly designed.

Occasionally Pascal's type checking would warn of a slip of the hand in writing a program; the run-time checking of values also indicated errors from time to time, particularly subscript range violations.

Turning to the negative side, recompiling a large program from scratch to change a single line of source is extremely tiresome; separate compilation, with or without type checking, is mandatory for large programs.

I derived little benefit from the fact that characters are part of Pascal and not part of Fortran, because the Pascal treatment of strings and non-graphics is so inadequate. In both languages, it is appallingly clumsy to initialize literal strings for tables of keywords, error messages, and the like.

The finished programs are in general about the same number of source lines as their Ratfor equivalents. At first this surprised me, since my preconception was that Pascal is a wordier and less expressive language. The real reason seems to be that Pascal permits arbitrary expressions in places like loop limits and subscripts where Fortran (that is, portable Fortran 66) does not, so some useless assignments can be eliminated; furthermore, the Ratfor programs declare functions while Pascal ones do not.

To close, let me summarize the main points in the case against Pascal.

1. Since the size of an array is part of its type, it is not possible to write general-purpose routines, that is, to deal with arrays of different sizes. In particular, string handling is very difficult.
2. The lack of static variables, initialization and a way to communicate non-hierarchically combine to destroy the ''locality'' of a program — variables require much more scope than they ought to.
3. The one-pass nature of the language forces procedures and functions to be presented in an unnatural order; the enforced separation of various declarations scatters program components that logically belong together.
4. The lack of separate compilation impedes the development of large programs and makes the use of libraries impossible.
5. The order of logical expression evaluation cannot be controlled, which leads to convoluted

code and extraneous variables.

6. The `case` statement is emasculated because there is no default clause.
7. The standard I/O is defective. There is no sensible provision for dealing with files or program arguments as part of the standard language, and no extension mechanism.
8. The language lacks most of the tools needed for assembling large programs, most notably file inclusion.
9. There is no escape.

This last point is perhaps the most important. The language is inadequate but circumscribed, because there is no way to escape its limitations. There are no casts to disable the type-checking when necessary. There is no way to replace the defective run-time environment with a sensible one, unless one controls the compiler that defines the ''standard procedures.'' The language is closed.

People who use Pascal for serious programming fall into a fatal trap. Because the language is so impotent, it must be extended. But each group extends Pascal in its own direction, to make it look like whatever language they really want. Extensions for separate compilation, Fortran-like COMMON, string data types, internal static variables, initialization, octal numbers, bit operators, etc., all add to the utility of the language for one group, but destroy its portability to others.

I feel that it is a mistake to use Pascal for anything much beyond its original target. In its pure form, Pascal is a toy language, suitable for teaching but not for real programming.

Acknowledgments

I am grateful to Al Aho, Al Feuer, Narain Gehani, Bob Martin, Doug McIlroy, Rob Pike, Dennis Ritchie, Chris Van Wyk and Charles Wetherell for helpful criticisms of earlier versions of this paper.

1. Feuer, A. R. and N. H. Gehani, ''A Comparison of the Programming Languages C and Pascal — Part I: Language Concepts,'' Bell Labs internal memorandum (September 1979).
2. N. H. Gehani and A. R. Feuer, ''A Comparison of the Programming Languages C and Pascal — Part II: Program Properties and Programming Domains,'' Bell Labs internal memorandum (February 1980).
3. P. Mateti, ''Pascal versus C: A Subjective Comparison,'' *Language Design and Programming Methodology Symposium*, Springer-Verlag, Sydney, Australia (September 1979).
4. A. Springer, ''A Comparison of Language C and Pascal,'' IBM Technical Report G320-2128, Cambridge Scientific Center (August 1979).
5. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Reading, Mass. (1976).
6. K. Jensen, *Pascal User Manual and Report,* Springer-Verlag (1978). (2nd edition.)
7. David V. Moffat, ''A Categorized Pascal Bibliography,'' *SIGPLAN Notices* **15**(10), pp. 63-75 (October 1980).
8. A. N. Habermann, ''Critical Comments on the Programming Language Pascal,'' *Acta Informatica* **3**, pp. 47-57 (1973).
9. O. Lecarme and P. Desjardins, ''More Comments on the Programming Language Pascal,'' *Acta Informatica* **4**, pp. 231-243 (1975).
10. H. J. Boom and E. DeJong, ''A Critical Comparison of Several Programming Language Implementations,'' *Software Practice and Experience* **10**(6), pp. 435-473 (June 1980).
11. N. Wirth, ''An Assessment of the Programming Language Pascal,'' *IEEE Transactions on Software Engineering* **SE-1**(2), pp. 192-198 (June, 1975).
12. O. Lecarme and P. Desjardins, *ibid*, p. 239.
13. A. M. Addyman, ''A Draft Proposal for Pascal,'' *SIGPLAN Notices* **15**(4), pp. 1-66 (April 1980).
14. J. Welsh, W. J. Sneeringer, and C. A. R. Hoare, ''Ambiguities and Insecurities in Pascal,'' *Software Practice and Experience* **7**, pp. 685-696 (1977).
15. N. Wirth, *ibid.*, p. 196.

16. J. D. Gannon and J. J. Horning, ‘‘Language Design for Programming Reliability,’’ *IEEE Trans. Software Engineering* **SE-1**(2), pp. 179-191 (June 1975).
17. J. D. Ichbiah, et al, ‘‘Rationale for the Design of the Ada Programming Language,’’ *SIGPLAN Notices* **14**(6) (June 1979).
18. J. Welsh, W. J. Sneeringer, and C. A. R. Hoare, *ibid*.
19. B. W. Kernighan and P. J. Plauger, *Software Tools in Pascal,* Addison-Wesley (1981).