ROBELLE CONSULTING LTD.
#130-5421 10th Avenue
Delta, B.C. V4M 3T9
CANADA
(604) 943-8021
Telex 04-352848

Experiences With Pascal

Technical Report, February 1981, By

DAVID J. GREER

Robelle Consulting Ltd.

Summary

This paper reviews the history and possible future of the Pascal programming language on the HP 3000 computer system. It reviews some of the currently available compilers, and discusses the features that are likely to be included in an HP-supported Pascal compiler. The presentation will cover practical problems encountered while using Pascal, including problems that arise in transporting Pascal programs from other machines to the HP 3000.

Contents

Introduction

This paper concentrates on one person's experience using and implementing the Pascal language on the HP 3000. The information presented was accumulated during approximately two years of work with Pascal on the HP 3000 and on an AMDAHL/V6 at the University of British Columbia. The major points covered by this paper are: 1) a brief history of Pascal, including implementation on the HP 3000; 2) the state of the contributed Pascal compilers; 3) an approximation of what the HP-supported Pascal may look like; 4) a look at Pascal in applications; what to avoid and what to be prepared for; and, 5) a preliminary evaluation of Pascal's performance.

History

Pascal was developed by Niklaus Wirth in Zurich. The principal aims of Pascal were to provide: 1) a language in which structured concepts could be taught easily; and, 2) a language that would be relatively easy to implement on many machines [18,15,1].

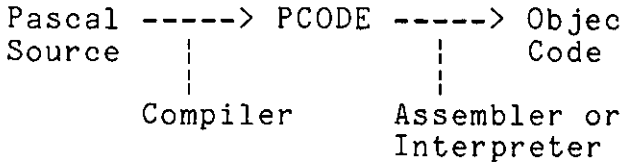
These original aims have direct application to business and scientific programming. The language provides constructs which emphasize structured coding concepts. Programs written in Pascal tend to be structured, which in turn makes them easier to maintain and understand. Because Standard Pascal has been implemented on many different machines, including the HP 3000, it is a good vehicle for writing portable software.

Another reason for the current interest in Pascal is that many secondary institutions, universities and colleges are now using Pascal as their principal programming language. As this trend continues, there will be increasingly more incentive to write software in Pascal, since the new work force will already be trained in the language.

Portable Pascal-P4

In order to make Pascal available on many machines, Urs Ammann, K. Nori, Ch. Jacobi, K. Jensen and H. Nageli wrote the portable Pascal-P4 compiler, which is itself written in Pascal [15]. Approximately 80% of the Pascal compilers in the world are based on this compiler.

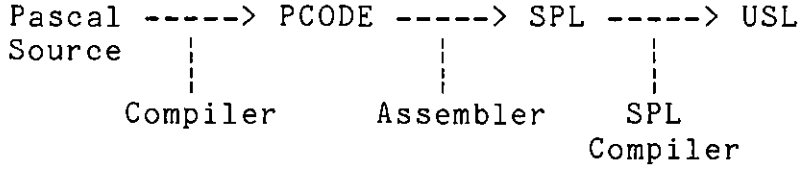
The P4 compiler takes Pascal source code and compiles it into symbolic code for a hypothetical stack machine called a "P-machine". The individual implementer of Pascal writes an assembler or interpreter for the "pcode". Later, the source program of the original compiler is changed to generate the host machine's object code directly. The following is a schematic description of how pcode works.



HP 3000 Pascal-P4

In the HP 3000 contributed library, there is a version of Pascal developed by Grant Munsey, Jeff Eastman and Bob Fraley. This compiler is a modified version of the Pascal-P4 compiler; it fixes several bugs in the original P4 compiler, and provides extensions to Standard Pascal. Some of the important extensions are: built-in procedures to do direct access to MPE files, an "otherwise" label in the case statement, calls to Pascal procedures which were compiled externally, and calls to procedures written in other languages.

The process of compiling Pascal programs on the HP 3000 is slightly different from the one described above. Instead of assembling pcode into object code, the assembler of the contributed version assembles pcode into SPL, which is then compiled into USL files.



While much of the work of implementing Pascal was simplified because of the P4 compiler, the early work done to transport Pascal to the HP 3000 was still difficult. Each of the P4 "pcode" instructions had to be translated into one or more SPL instructions. Also, the basic Pascal-P4 compiler only allowed for character constants of ten characters or less - a severe restriction [5]. See Appendix II for more details.

Pascal-V

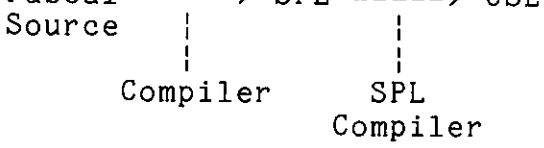
History

Pascal-V is another version of Pascal for the HP 3000, developed in Vancouver as a project in compiler design at the University of British Columbia. This compiler is a modified version of the one that is available in the contributed library [3,4,5].

Major Problems

Two of the problems in using the contributed library's Pascal are: it is somewhat difficult to specify all of the command sequences for invoking the compiler (although this has been fixed with UDCs), and, the compiler is very slow. Both of these problems stem from the long process required to get from Pascal source code to USL files.

Pascal-V does away with the assembly stage of the compilation process. Instead, Pascal source code is translated directly into SPL code. The Pascal compiler then invokes the SPL compiler to produce the USL file.



By eliminating the pcode stage of the process, a 20-40% savings in elapsed compilation time was realized. Also, by having all of the compilation stage in the compiler, it was easier to use the compiler. The SPL stage of the compilation process was not eliminated, because it was too difficult to work with USL files directly.

Minor Problems

The original contributed compiler did not print error messages. The compiler now prints a summary of all of the error numbers which occurred during compilation, along with their associated error messages. In addition, a compiler option has been provided which causes all Pascal reserved words to be underlined. This greatly enhances the readability of Pascal programs.

Athena Compatibility

With the Athena MIT release of MPE (2011), all Pascal programs on the HP 3000 ceased to work, including all of the contributed versions of Pascal. The cause of this was that, as of the Athena release of MPE, Qinitial (the initial setting of the Q-register in the program's data stack) was two words higher in the stack. Since Pascal made certain assumptions about where Qinitial would be in the stack, Pascal programs stopped working.

Pascal-V fixes this bug by making no assumption about where Qinitial should be.

Standard Pascal

There is a world-wide effort to provide a comprehensive standard for Pascal. Any compiler which claims to compile Standard Pascal must compile all parts of the defined standard correctly. In order to help implementers and users of Pascal find out whether their particular compiler meets the standard, A. Sale and R. Freak have written a suite of Pascal programs to test Pascal compilers.

The suite consists of approximately 300 Pascal programs. Each program is compiled and executed by the Pascal compiler. The results of each program are analyzed for errors. If a compiler meets Standard Pascal completely, there will be no errors from any program in the suite.

Pascal 2.4-V was tested using this suite. It had as many errors as other compilers based on the original P4 compiler (that is, the compiler was average). Several of these errors were fixed in Pascal 2.5-V and later versions.

Usage

The Vancouver version of Pascal is currently used in about forty installations around the world. Many installations are also using Pascal-S (see Appendix I) for teaching Pascal. Two installations are using Pascal to convert software from other machines to the HP 3000.

Availability

Because we need a Pascal compiler now to develop programs at Robelle Consulting, I have been and will be maintaining this Pascal compiler. It is not a supported Robelle product; but, for $200 U.S. (to defray costs), we will send a magnetic tape copy of the latest version to interested users ($100 if you send payment with your order and do not ask for 800 BPI). As of January 1981, the latest release was Version 2.8-V. A bug was fixed that did not allow compiles while logged on as MANAGER.SYS. Enhancements were made in error messages on file opens, in the use of Control-Y and in the run-time support (so that all Pascal programs can read QEDIT-format files). Inquiries can be directed to me at Robelle Consulting Ltd., #130-5421 10th Ave., Delta, B.C., V4M 3T9, Canada. Phone: (604) 943-8021. Telex: 04-352848.

Future Enhancements

Three major problems (as well as many minor ones) remain to be fixed in the compiler, if it is to be used in a commercial environment. The first is that character strings are stored as one character per word, rather than one character per byte. This needs to be fixed, so that variables will use less memory space, and so that Pascal programs can communicate directly with HP

subsystems such as IMAGE.

The second problem has to do with Pascal's definition of parameters to procedures and SPL's definition of parameters to procedures. SPL is known as a programming language with weak type checking. This gives the programmer more flexibility, but provides more opportunity for making mistakes. It also allows IMAGE parameters to be defined very loosely. For example, a data set can be defined by a character-string containing the name of the data set, or by an integer variable with the set number. Since Pascal does not allow such flexibility in parameter passing, some mechanism must be established to allow procedure calls to subsystems such as IMAGE.

The third problem is that Pascal integers are implemented as single-word integers. Many subsystems such as IMAGE require that double-word integers be passed as parameters. At the same time, there must be a way of declaring single-word integers, since other subsystems require both single- and double-word integers to be passed as procedure parameters.

Proposed HP Pascal

Recently, there have been several rumors that HP would provide a supported Pascal compiler for the HP 3000 some time in the future. The following is an educated guess as to what this compiler may look like when, and if, it arrives.

The HP compiler is to be based on an internal HP standard for Pascal. The HP 1000 Pascal compiler, which was recently introduced, is also supposed to follow the internal HP Pascal standard. Any comments I make about Pascal for the HP 3000 are based on how things are done on the HP 1000 [9].

If Pascal/3000 looks very similar to Pascal/1000, we can look forward to an excellent implementation of the language. Pascal/1000 provides features which take advantage of the HP 1000 operating system, yet still retain the "spwirit" of Pascal. Of special importance is the inclusion of a compiler option which permits the compilation of Standard Pascal only. By turning this option on, only Pascal source that followed the standard would compile.

Storage allocation in Pascal/1000 is done in a very flexible manner. Two restrictions of the contributed versions of Pascal are that neither double word-integers, nor sets with greater than 62 elements are allowed. Pascal/1000 permits sets with up to 32767 elements, and only allocates as much storage as is necessary. Similarly, both single- and double-word integers may be declared in a way that is natural for the Pascal language.

Pascal/1000 also allows character strings to be stored as one character per byte, instead of just one character per word. Procedure calls are allowed to external procedures written in either Pascal or HP 1000 assembler.

Assuming Pascal/3000 will follow the lead of Pascal/1000, it should be a very successful compiler. The only problem to which I've no solution is how Pascal/3000 will permit calls to HP subsystems like IMAGE, while working within the confines of Pascal type checking.

Pascal in Applications

This section attempts to describe some of the common pitfalls to watch for when using Pascal. It also does a step by step examination of each of the Pascal types, in the context of their use in future versions of Pascal and with other subsystems. For those just learning Pascal, [18,8,17] may be useful. In particular, [8] gives a complete and readable treatment of Pascal.

Pascal Types

The concept of types in Pascal is one of the most powerful features of the language. Because so much of Pascal operates from the concept of types, it is one of the main areas where problems can occur, especially when dealing with different Pascal compilers.

Integer

The integer type is one of the simplest and most common types in Pascal. The defintion of integer is as follows:

```
type

    integer = -maxint .. +maxint;
```

The notation '-maxint .. +maxint' is called a subrange; it defines integer to be a type that can take on values from a lower bound (-maxint) to a higher bound(+maxint). The first question that one usually asks is just how large is maxint? The answer is that it varies from compiler to compiler. In Pascal-V it is +32767, but in the future it will likely be +2147483647. In the first case, storage will be allocated as a single- word integer, and in the second case, storage will be allocated as a double-word integer.

Suppose that the compiler used the second value for maxint. How could you declare a type that only used single-word storage? It could be done as follows:

```
type

    int = -32767 .. 32767;
```

If the compiler is "smart" in storage allocation, int would only use single-word storage. The int declaration also has the advantage of telling the reader exactly what range any variables of type int should have.

Real

The problem of a value range for real numbers (and the amount of storage to allocate) is similar to the problem of single and double integers. The main difference is that reals cannot be used in subrange notation. The storage allocation and size attributes of reals are left totally up to the implementer of each Pascal

compiler.

Since there are many instances where different size reals are needed, there is a general solution. But, this solution is not part of Standard Pascal. The predefined type "Real" is usually taken to mean single precision floating-point numbers, where the size of single precision floating-point numbers is defined for each host machine. On the HP 3000, it is a 32-bit number with approximately seven decimal positions. For larger real numbers, the predefined type "Longreal" is provided, which is usually taken to mean double precision floating-point numbers. On the HP 3000, these would be represented by 64-bit long-real quantities. Currently, all of the HP 3000 Pascal compilers supply only the type Real. Since future compilers will probably allow for larger real numbers, the following declaration could be used to localize the changes at a later date:

```
type

    Longreal = Real;
```

Whenever "Longreal" became available, this type declaration could be deleted, and the program would just need to be recompiled.

Boolean

Logical values in Pascal are represented by True(1) and False(0). On the HP 3000, Boolean is implemented as a single-word integer. The Pascal-V compiler checks for False = 0 (True = not 0) when examining the value of a boolean expression; but, this may change, and shouldn't be counted on.

Characters

One of the main reasons more people are not using Pascal on the HP 3000 is that characters are packed one per word, instead of the regular one per byte. This means that a Pascal array[1..n] of char cannot be passed to procedures in other languages, without first packing the character array.

The Pascal type packed array[1..n] of char will someday have characters packed one per byte. For this reason, packed should be used wherever possible. It also requires less storage. One word of caution: it is very likely that packed types will not be able to be passed as var parameters to a procedure or function. To change a packed array to an unpacked array, the built-in procedure unpack should be used, and the built-in procedure pack should be used to convert in the other direction.

Set

Sets are one of the more unique concepts available in Pascal. By using sets, it is possible to have a single variable take on more than one value at the same time. This could be very useful in certain application areas (for example, on a customer status field, where a customer could be in two different status classes

simultaneously).

Again, there are few guidelines regarding the implementation of sets. The limiting factor in declaring sets is the number of distinct elements that can be in one set type. With Pascal-V, there can be up to 62 elements in a set. Unfortunately, this rules out the following useful type:

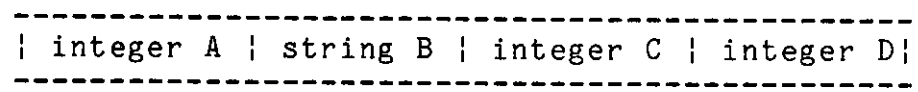
```
type

     charset = set of char;
```

With Pascal-V, set types occupy four words of storage, where each bit in the four words represents one of the values in the base type. Other versions of Pascal are more likely to allocate as many words as necessary to represent the base type. This means that writing out set types to files, or calling procedures in other languages with parameters of type set is very unwise. At the very least, such calls should be isolated so that they can be changed easily.

Many implementations of Pascal allow only 48 distinct elements in set types. This should concern anyone who intends to write Pascal software for other machines. For portable software, you should use sets with a small number of elements. While this is unfortunate, it is likely to be the case for many years.

Record

Record structures allow similar things to be grouped together and optionally given a name. This feature is similar to the COBOL level structure. With COBOL, a level structure is allocated space in the order that the various levels are allocated. If we were to map the following record structure into COBOL it would look like this:




```
01  RECORD.
    05  A                       PIC S9(4) COMP.
    05  B                       PIC X(10).
    05  C                       PIC S9(4) COMP.
    05  D                       PIC S9(4) COMP.
```

And the equivalent structure in Pascal-V would be:

```
type

     int    = -32767 ..  +32767;
     string = packed array[1..10] of char;

     cobol_record = record
                        d : int;
                        c : int;
```

```
        b : string;
        a : int;
    end;
```

Note that the Pascal record structure is exactly opposite to what you would expect. This is because Pascal-V allocates storage elements in reverse order to their declaration. This is implementation-defined; other Pascal compilers may do exactly the opposite, or even something in-between. For this reason, all of your record structures should be declared in one place, using type and $include filee (this facilitates changes, when necessary).

IMAGE/COBOL/Pascal Table

The following table gives equivalences among IMAGE, COBOL and Pascal types. [10] The table assumes the following Pascal types are available:

```
type

    int      = -32767 ..  32767;
    integer = -2147483647 ..  2147483647;
    real     = real;      (* single precision floating-point *)
    longreal= longreal; (* double precision floating-point *)
```

IMAGE	COBOL	Pascal
J1	PIC S9(4) COMP	int
J2	PIC S9(9) COMP	integer
I1	PIC S9(4) COMP*	int
I2	PIC S9(9) COMP*	integer
K1	PIC 9(4) COMP*	boolean*
K2	PIC 9(9) COMP*	integer*
P4	PIC S9(3) COMP-3	packed array[1..2] of char*
P8	PIC S9(7) COMP-3	packed array[1..4] of char*
R2	PIC X(4)*	real
R4	PIC X(8)*	longreal
Z N	PIC S9(N)	packed array[1..N] of char*
X N	PIC X(N)	packed array[1..N] of char

* - Storage is allocated correctly, but the types do not really correspond to the IMAGE types.

Note that the concept of packed decimal does not exist in Pascal. The only way to handle packed type data is to have a set of SPL procedures which do the conversion from packed decimal to some internal format, and the reverse, as well as providing for the actual arithmetic. The most likely data type to hold packed decimal numbers would be a packed array[1..N] of char to hold the packed decimal numbers.

The COBOL zoned-decimal type is not supported directly in Pascal. While the numbers can be read in as a packed array, they will have to be converted to integer by hand. The last byte of the zoned-decimal array will contain the digit value, as well as the sign. For some sample solutions to these problems see [6,7].

Files

In general, the only type of file that is compatible between Pascal and other HP 3000 languages is:

```
type

    filetype = file of array[1..n] of char;
```

All other file-types are likely to be incompatible, or at best, difficult to interpret. While it is certainly possible to declare a file of the type cobol record above, the results are not what one expects. In the first place, all of the elements of the record will be reversed, so that the COBOL and Pascal record layouts for the file must be reversed. Further, the string that is part of the record will be packed one character per word with Pascal-V. (Pascal does not currently pack strings, even if packed is included in the declaration.) In order to reduce difficulties with program maintenance, it is recommended that there be one common input and output interface to an external file from Pascal programs. These routines should be included in any program that is to use the file.

IMAGE

All of the problems mentioned above apply even more to IMAGE and Pascal. The layout of a data set and the layout of the Pascal record must be reversed with Pascal-V. In addition, there are currently no double-word integers or packed arrays, so calling IMAGE procedures would be difficult.

As mentioned above, the definition of IMAGE procedures does not agree with that of Pascal procedures. Until a Pascal compiler on the HP 3000 recognizes all of the HP intrinsics, it will be necessary to code calls to IMAGE in some other way. The suggested solution is to provide one SPL procedure for each data set that is to be used in Pascal.

Each SPL procedure would have a mode parameter (read, chainread, write, update, etc.), as well as the other IMAGE parameters (the base name, the status area and a buffer). It would make sense for the SPL interface to use the "@" list in all DBGET calls (as well as assuming the set name). Each interface routine would transform the IMAGE record structure into the equivalent Pascal structure (i.e., unpack character arrays, reverse the order of records, etc.). This is a general solution which should work for different versions and implementations of Pascal on the HP 3000.

Performance of Pascal Programs

Introduction

One of the most frequently-asked questions about the implementation of a programming language is: how fast are the programs it generates? On a machine like the HP 3000, where many factors contribute to the performance of a program, it is a difficult question to answer.

Programs written in the host programming language are generally the fastest, because the host programming language was custom-designed for the particular machine. Most of the constructs in the language translate directly into hardware instructions. This is true of SPL on the HP 3000. Other languages (like Pascal or COBOL, designed for use on many different machines), must translate the language constructs into a combination of hardware instructions and procedure calls to a run-time library. Since the procedures in the run-time library are generally slower than hardware instructions, there are certain language constructs to watch out for.

What to Watch Out For

Not all high-level language constructs are slow. Many of the language features will be as fast as a lower-level language like SPL. On any particular machine there are certain things to watch out for and avoid, because they are unusually slow. Pascal is no exception to this rule.

One of the most common operations is to open a file and read it sequentially from beginning to end. In Pascal, this is often accomplished by using the built-in procedures reset and read. The reset procedure opens the file, and the read procedure reads the file as if it were one long string of characters. Since files are normally organized into records, it seems obvious that reading the file one character at a time will be inefficient.

An Example

Pascal provides another built-in procedure, get, which will read a file one record at a time, if the file is declared correctly in the Pascal program. In order to test the time necessary to do a sequential read, the following Pascal declaration was made:

```
type

    filetype = file of array[1..80] of char;
```

The file input was declared to be of this type, and get was used to read the file sequentially. Figure I and II give a graphic demonstration of the elapsed and CPU time of four programs which read a sequential file.

Elasped Time vs. Number of Records Read

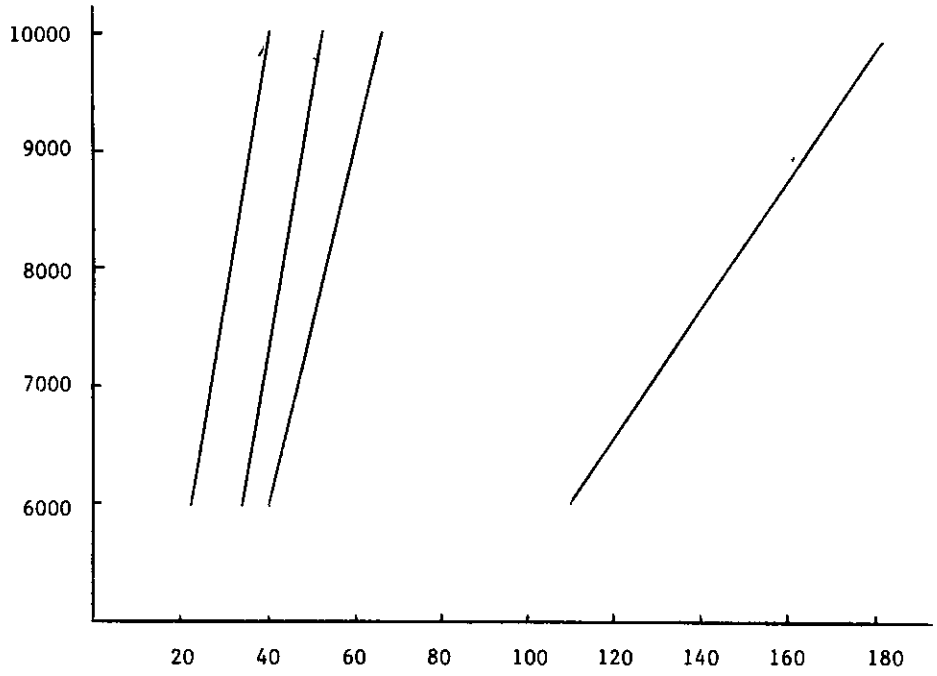


Figure I

CPU Time vs. Number of Records Read

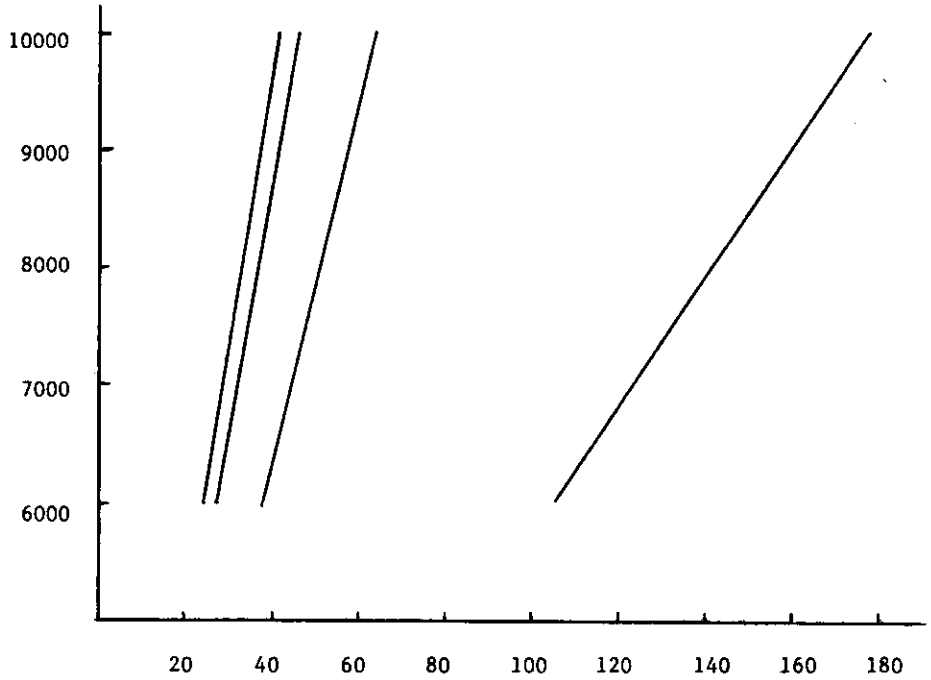


Figure II

One program was coded in SPL, using the FREAD intrinsic. Another was coded using the COBOL read statement. One Pascal program was coded using get, and another was coded using read. As expected, the SPL program was the fastest, followed by the COBOL program. The Pascal program using get was slower than the COBOL program, but substantially faster than the Pascal program using read.

Explanation of the Results

SPL was the fastest, because it communicates with the operating system directly, doing a FREAD of 80 bytes and checking the condition code for end-of-file.

The COBOL program interfaces to FREAD through the COBOL run-time support. The run-time support is general-purpose; it must handle all different file types and file sizes and it also does more error checking. The extra time consumed in the run-time support routines makes the COBOL program slightly slower than the SPL program.

The Pascal program suffers all of the same problems as the COBOL program. The Pascal run-time support must be prepared for all situations. In addition, the buffer that is read for each record must be unpacked, since the current version of Pascal does not support packed files. The extra time to do the unpacking shows up dramatically in Figure II, where there is a large difference in CPU time between the COBOL program and the Pascal program using get.

Finally, the Pascal program using read was much slower than any of the other programs. The reason for this is that an extra procedure call is made for every single character in the file. All of this extra overhead results in a Pascal read being much slower than a Pascal get. (The only thing that could be slower is the FORTRAN formatter with a format of 80A1, which calls the formatter for each character).

The Moral

Each implementation of a high-level programming language has certain constructs which are inefficient. For example, COBOL programmers use COMP variables when the value to be represented is less than ten digits long, and they avoid the use of the COMPUTE verb, since it is very slow in COBOL68. Pascal programmers should be aware of certain Pascal constructs that are slow on the HP 3000. In particular, this example shows that Pascal get should be preferred over Pascal read, when sequentailly reading a file. See Appendix II.

Appendix I

Making Pascal Portable - Three Examples

I. Pascal-S

Pascal-S is a subset of standard Pascal. [18,19] The compiler itself is a complete system which compiles the Pascal program, and, if there are no errors, executes the program (interpretively). Pascal-S was written in Pascal by Wirth.

The major problem in getting Pascal-S running on the HP 3000 was the difference between the character set on the CDC series of machines and the character set of the HP 3000 (ASCII). In the CDC character set, blanks collate after letters; but, in the ASCII character set, blanks collate before letters. Also, the internal numeric representation of characters differs between the CDC and the HP 3000, and this caused further errors in the compiler.

The lesson to learn from this is that writing portable programs, even in Pascal, takes some thought and effort. Even when using Standard Pascal, problems can arise. One tip: make no assumptions about character sets, as they vary widely from machine to machine.

II. PROSE

PROSE is a text formatter published in Pascal News No. 15 [16]. It is written entirely in Standard Pascal and pays particular attention to the character sets of different machines. PROSE stores all text internally using the ASCII conventions, and each implementation (of PROSE) must have routines to convert from the external character set to the internal one.

PROSE is approximately 3500 lines long, and is just now being completed on the HP 3000. The main problems encountered in implementing PROSE were finding and eliminating typing mistakes, and understanding the conversion from the external character sets to the internal set. Even though the external and internal character sets are the same on the HP 3000, it took some time to find and change the conversion routines accurately. The coding of these routines assumed an understanding of how the CDC character sets work, since the version of PROSE written in Pascal News No. 15 was for a CDC computer.

Another problem encountered with PROSE was the definition of carriage control on output files. Many implementations of Pascal take the first character written to each line of an output file as the carriage control character. For example, to write a page eject, the following Pascal code would be written:

```
writeln; writeln('1');
```

The "1" in the second writeln would be interpreted as a carriage control character, which, on most line printers, causes a page eject. Pascal 2.8-V uses standard built-in procedures to do

carriage control. The procedure page(output) causes a page eject on the file output. Because PROSE was implemented using the first method, it was necessary to change each carriage-control write statement to a similar or equivalent Pascal 2.8-V statement. While this is relatively straightforward with page ejects, it is much more difficult with overprint and underlining.

The lesson to learn from this is that the meaning of carriage control on output files is implementation-defined. In order to make a Pascal program portable, all carriage control should be done by a few, easy-to-modify procedures, rather then by any implementation features. These procedures can then be modified for each Pascal installation.

III. LISP

LISP is a small implementation of the interpretive language LISP. It was written at the University of British Columbia using Pascal/UBC, an extended version of Pascal. There were only two major problems in getting LISP to work on the HP 3000.

The first was that Pascal/UBC runs on an AMDAHL/V6, using the EBCDIC character code. On most IBM terminals that use EBCDIC, there are no "]", "]" or "^" characters. Because of this, Pascal/UBC accepts "(." as "[", ".)" as "]" and "@" as "^". Each of these character sequences had to be converted to their ASCII counterparts.

The second problem centered around the use of the Pascal forward declaration. Since LISP is essentially recursive in nature, all of the procedures of LISP were declared forward to avoid the mutually recursive problem. The use of forward in Pascal is not clearly defined; but most implementations of Pascal require that the parameter list to a forward procedure or function be declared when the procedure or function is delcared forward, and that the actual parameter list be left off when the procedure or function is actually declared.

Pascal/UBC permits an extension whereby the parameter list may be declared both when the procedure or function is declared forward and when the actual procedure or function body is declared. Pascal 2.8-V does not allow this extension, so each actual declaration of a procedure or function had to have the parameter list deleted.

The lesson to learn from this is that non-standard Pascal features must be avoided, if a Pascal program is to be made portable. Most Pascal compilers have a compiler option which permits only standard Pascal to be compiled. Before declaring a Pascal program portable, be certain to turn the standard compiler option on, recompile and rerun the program, to ensure that it is free from any non-standard Pascal features.

Appendix II

Implementation Notes on Pascal-V

Introduction

This appendix is intended for the interested reader who wishes to know more about the actual implementation of Pascal. It assumes that the reader is already familar with Pascal, the HP 3000 operating system, SPL/3000, and the HP 3000 instruction set [18,15,5,1,11,12,13].For those interested in compiler design in general, [20] gives a readable introduction to recursive-descent compiling, and [2] covers the general topic of compiler design thoroughly.

Basic Compiler Design

The compiler itself is written in Pascal. It uses recursive-descent compiling techniques. The original compiler compiled code for a hypothetical stack machine [15]. The principal modules and their functions are as follows:

nextsym - reads the input text and returns the next lexical token. This module is also responsible for output, including error messages, and all input, as well as $INCLUDE files. If an identifier is encountered, it also looks up the identifier to see if it is a reserved word, but it does not check to see if the identifier was already defined by the user.

table management - these routines store all user-defined identifiers into the various compiler tables, and provide lookup of identifiers from the same tables. Table storage is organized as an unbalanced binary tree for each program level [20,14].

block - this is the syntax recognizer of the compiler. It also generates the "pcode" pseudo-instructions, and handles all parsing and semantic definition. The structure of block is broken down as follows:

```
label                          label-declaration-part
const                          constant-declaration-part
type                           type-declaration-part
var                            variable-declaration-part
procedure-and-function-declaration-part
statement-part
```

The statement-part handles all of the various statements available in Pascal. The structure of block follows the syntax diagrams in the User Manual and Report [18].

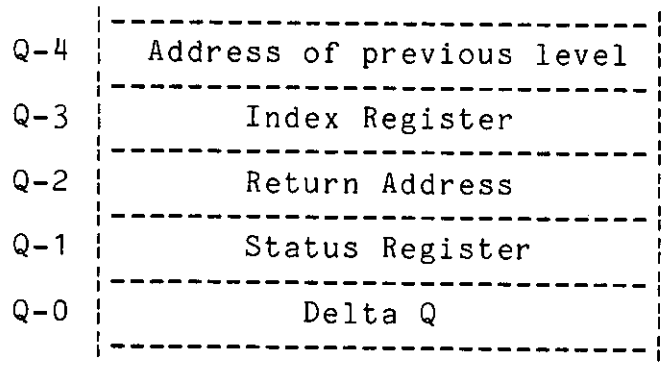
initialization - this module handles all static and dynamic initialization. This includes the reserved words, input/output and nextsym variables, as well as the predeclared types and procedures. Predefined procedures and types are stored just like regular user identifiers, so that they may be overridden by the Pascal programmer.

SPL code generation - these procedures translate the "pcode" statements into valid SPL statements. A combination of regular SPL, ASSEMBLE and TOS is used in the translation to SPL. Note that the basic code generation is still based on the "pcode" machine. However, in Pascal-V the "pcode" translation stage, which generated an external file and ran the program ASSM, has been eliminated. The code that is generated still has some of the basic deficiencies of the "pcode" machine.

Stack Frames

Usually, when implementing Pascal, a major problem is the provision of the procedure call and return mechanism. Fortunately, most of the tools required are already available on the HP 3000, in the form of the PCAL and RETURN statements of the HP 3000 instruction set.

When a procedure is called using PCAL, a four-word stack marker is loaded onto the top of the stack before control is transferred to the new procedure. In SPL, this stack marker is sufficient to save and restore the entire SPL environment. However, SPL provides only single-level addressing, while Pascal provides multiple-level addressing. To implement "up-level" addressing, each Pascal procedure call loads a five-word "stack frame" onto the top of the stack, instead of a four-word stack marker. The stack frame is structured as follows:



This is the Pascal stack frame. Words Q-3 through Q-0 are loaded automatically by the PCAL instruction. The DB-relative address of Q-0 of the previous level is placed onto the top of the stack by the Pascal compiler, before the PCAL is executed.

Since all procedure levels are established at compile time, it is possible to compute the address of a variable that is neither local nor global. The Pascal program walks back through the correct number of stack frames to reach the beginning of the level where the variable to be used is located. Then the Pascal program works forward from the computed address to obtain the actual address of the variable in question.

Parameter Passing

Most parameters are passed to Pascal procedures in the same

way that parameters are passed in SPL. For reference parameters, the word address of the parameter is loaded onto the top of the stack before the procedure is called. Once the procedure is called, it uses Q-indirect addressing to obtain the value of the reference parameter.

Simple scalar value parameters are also passed as in SPL. The actual value of the parameter is loaded onto the top of the stack before the procedure is called. Set, record and array value parameters are permitted in Pascal, but not in SPL. The principle remains the same: space is reserved on top of the stack for the value parameter, then the actual value of the set, record or array is copied into the reserved space. Once the procedure is called, it uses Q-negative addressing to obtain the value of any one of the variables.

Each Pascal procedure must know exactly how many words of storage are taken by both its value and reference parameters. When the Pascal procedure returns to the invoking routine, it deletes all of its parameters from the stack.

Function Return

The last problem involved in calling Pascal procedures or functions is where to leave the result of a function when it returns. The calling routine reserves enough space for the result on top of the stack, before loading the parameters or the stack frame.

Since only scalar, subrange or pointer types may be returned as functions, the compiler reserves either one or two words on top of the stack for the function result. Two words in the case of reals, and a single word in all other cases. Once the function is called, it uses Q-negative addressing to store the function result. Upon return, the space for the result is the only space not deleted from the stack. Therefore, the result is always on top of the stack after a function call.

Addressing

In general, Pascal uses DB+ addressing for global variables, Q+ addressing for variables declared at the current level, and Q-negative addressing for procedure or function parameters. When a variable from another level, other than global, is needed, the X register is used.

One problem encountered in Pascal that is not present in SPL is that Q-relative addressing is only allowed from Q-63 to Q+127. Since value parameters of unlimited size can be passed in Pascal, and it is easy to declare a record structure that is more than 128 words long, provision had to be made for larger Q-relative addresses. In order to work around this problem, Pascal uses a combination of the Q register and the X register.

Whenever a reference is made to a local variable whose address is larger than Q+127, the following algorithm is used.

The nearest multiple of 128 is stored in the X register. The difference between the address of the variable and the value in the X register is then used as the Q-relative address. Any reference instruction then uses combined Q-relative and indexed addressing. Q-negative addressing is similar, except that the value stored in the X register is a multiple of 64, because Q-negative addressing only allows for addresses up to Q-63.

For example, take a variable whose address would be Q+130, and assume that a single-word load to the top of the stack was to be done. The following SPL code would do the actual load:

```
X := 128;                  <<CLOSEST MULTIPLE OF 128>>
ASSEMBLE(LOAD Q+2,X);      <<130-128 = 2>>
```

A similar situation exists with DB-relative addressing, but the limit is DB+255. The same solution is used as in Q-relative addressing, except that the closest multiple of 256 is loaded into the X register, and the difference between the desired location and the X register is used as the DB-relative address. A combination of DB-relative and indexed addressing is then used to reference the global variable.

Why Didn't Pascal Work Under Athena?

The contributed Pascal compilers would not run with the Athena (2011) release of MPE, because Qinitial was two words higher in the stack. If the Pascal compiler used DB-relative addressing for all of its global variables, why would it matter where Qinitial was?

The answer to this question is that IF Pascal always used DB-relative addressing for global variables, it would not matter. Unfortunately, over time, and through various changes to the compiler, some references to global variables became Q-relative. Since the structure of the initial Pascal stack was precisely defined, it did not matter whether DB-relative or Q-relative addressing were used for global variables, so long as the addresses were computed properly at compile time. As long as Qinitial was always exactly four words higher in the stack than secondary DB, there was no problem.

With the Athena MIT of MPE, Qinitial was six words above secondary DB (or more, if INFO= was used in the :RUN command). But Pascal continued to use Q-relative addressing on some global variables, and, under Athena, these were not the correct variables. The fix to the compiler consisted of tracking down every global reference and ensuring that it used DB-relative addressing. Once this was done, Pascal was no longer concerned with the position of Qinitial, and all Pascal programs could run on any release of MPE.

Dynamic Memory-Allocation

One of the most powerful Pascal features, for both applications and teaching, is the concept of pointer-variables and

dynamic memory-allocation. Whenever the built-in Pascal procedure new is used, storage must be allocated for the new variable. The amount of storage allocated is determined by the type of the object passed as a parameter to new.

Pascal maintains an area called the heap. The heap is an area of memory, logically separate from the "stack", which can be used for dynamic memory-allocation. On the HP 3000, the heap is implemented as the DL area. A dynamic counter of the current size of the heap is maintained in every Pascal program. When the procedure new is used, the counter is incremented, and the resulting address is stored in the variable passed to new. If the counter has passed the current limits of the DL area, the DL area is expanded by 1024 words (using the DLSIZE intrinsic).

In order to give the Pascal programmer some control over the size of the heap, two built-in procedures, mark and release, are provided. Mark stores the current size of the DL area in the variable passed to mark. This same variable is passed to the procedure release, which shrinks the DL area back to that original size. If the value of the variable used to mark the heap is changed before the call to release, the DL area will be shrunk by an unpredictable amount.

Run-Time Libarary

The run-time library gives Pascal significant power to deal with files, and other features of the HP 3000 which are external to the Pascal program. From a Pascal program, it is a simple task to write out an integer, real or character value, especially when compared with the effort needed to do the same thing in SPL. The run-time support allows Pascal this "friendly" type of communication with files.

The run-time library consists of approximately 1800 lines of SPL code. The main entry points to the run-time library, and their functions, are listed below:

PASCAL'FERR	Writes out Pascal file error messages.
PASCAL'ERROR	Writes out general Pascal run-time errors.
PASCAL'CLOSE	Closes an open Pascal file.
PASCAL'CLOIO	Closes the standard files INPUT and OUTPUT.
PASCAL'OPEN	Opens a Pascal file for either read or write.
PASCAL'GET	Gets the next record from the file. This procedure is called when the built-in procedure get is used.
PASCAL'GCH	Returns the next character in a file buffer.
PASCAL'PUT	Writes out the current file record to the file. This procedure is called when the built-in procedure put is used.
PASCAL'PCH	Adds a character to the output file buffer.
PASCAL'POS	Positions a file to a particular position.
PASCAL'FILE'POS	Returns the current file position.
PASCAL'RESET	Equivalent to the built-in procedure reset.
PASCAL'REWRITE	Equivalent to the built-in procedure rewrite.
PASCAL'CY	Pascal Control-Y trap.

PASCAL'GETHEAP	Expands the DL area by 1024 words.
PASCAL'INIT	Initializes the Pascal environment and opens the files INPUT and OUTPUT.
PASCAL'RDI	Reads an integer from a text file.
PASCAL'RDR	Reads a real from a text file.
PASCAL'WRI	Writes an integer to a text file.
PASCAL'WRR	Writes a real to a text file.
PASCAL'WRS	Writes out a string to a text file.
PASCAL'WRB	Writes out a boolean value to a text file.
PASCAL'DATE	Obtains and formats today's date.
PASCAL'TIME	Obtains and formats today's time.

These functions are stored in the RL file and are copied into the Pascal program by means of the RL= parameter of the :PREP command.

File Buffers

Each of the file-handling procedures above is passed a pointer to an array which acts as the Pascal file-control block. The structure of the file-control block is as follows:

Word	Use
1	MPE File Number
2	Current Record Length of the Buffer
3	Maximum Record Length of the File
4	Control Bits | Carriage Control
5	Length of the Buffer
6	Current Character Position in the Buffer
7	Current Character If TEXT File
8	Next QEDIT Block
9	Next QEDIT Index
10	QEDIT File Flags
11	QEDIT Linenumber of Current Line
12	Variable Length Buffer Size depends on the type of file. For TEXT files, this buffer is 128 words long.

When a Pascal program uses a read statement, characters are not obtained one at a time. Instead, a buffer containing the the line most recently read from the file is used, along with a character position in the buffer. If all of the characters in the buffer are read, the next line of the file is read automatically. Since only 256 bytes are reserved for the file buffer of a text file, this is the largest record size that the actual MPE file attached to the text file may have.

When a file is declared in a Pascal program, the local storage for the file-control block is allocated at the point where the file is declared. This method doesn't work for the standard files input and output, since they are predefined automatically in every Pascal program. To get around this problem, the file-control blocks for the standard files input and output are declared in the DL area. When the Pascal program is first run, these file-control blocks are allocated and initialized. One of the reasons that a Pascal procedure cannot be called from another language is that this allocation and initialization of the input and output files would not be done properly. If the Pascal program does any reading from input or any writing to output, then it would fail, due to the lack of the correct file-control blocks.

Summary

The only software more complicated than compilers are operating systems; yet compilers are vital to our continued use of computers. Pascal-V is approximately 7500 lines of Pascal, and it has been modified by at least three different people. Despite this, many useful programs have been developed using Pascal-V. The Pascal-V compiler is a reasonably solid and reliable system, which provides a good base for incremental enhancements in the future.

Appendix III

Sample Data From the Sequential Read Test

I. Elasped Times

No. Records	SPL	COBOL	Pascal/Get	Pascal/Read
6000	26.34	34.41	39.77	109.44
7000	30.69	40.13	46.31	127.55
8000	35.01	45.78	52.87	145.79
9000	39.36	51.50	59.44	164.03
10000	43.70	57.20	65.99	182.15
y-int	-70.97	-42.82	-68.36	-14.94
slope	230.47	175.59	152.60	54.97
Corr. Coeff.	0.9999	0.9999	0.9999	0.9999

II. CPU Times

No. Records	SPL	COBOL	Pascal/Get	Pascal/Read
6000	25.34	28.03	38.62	108.03
7000	29.56	32.69	45.01	125.98
8000	33.78	37.35	51.48	144.03
9000	38.00	42.01	57.93	162.11
10000	42.21	46.66	64.35	180.04
y-int	-8.06	-18.03	4.06	4.56
slope	237.08	214.68	155.33	55.50
Corr. Coeff.	0.9999	0.9999	0.9999	0.9999

All of the data was compared using a least squares fit of a linear line. The resulting slopes and y-intercepts are listed. Notice that all programs had a near perfect correlation coefficient; this indicates that the file system has a linear increase in time as the number of records sequentially read increases.

Bibliography

[1] Addyman, A. M.
"A Draft Proposal for Pascal"
SIGPLAN Notices Vol. 15 No. 4, April 1980
Association for Computing Machinery,
1133 Avenue of the Americas, New York, NY 10036

[2] Aho, Alfred; Ullman, Jeffrey
Principles of Compiler Design
Addison-Wesley, Don Mills, Canada, 1977

[3] Earls, John
"Pascal for the HP 3000"
HPGSUG Journal, Vol. 1, No. 5

[4] Fraley, Robert
"Pascal-P on the HP 3000"
HPGSUG 1980, San Jose Proceeedings

[5] Fraley, Robert
"The Pascal Programming Language"
HPGSUG 1980, San Jose Proceeedings

[6] Green, Robert M.
SPL Aids, Sofware Package
Robelle Consulting Ltd.

[7] Green, Robert M.
SPL/3000 in a Commercial Installation
Robelle Consulting Ltd.

[8] Grogono, Peter
Programming in Pascal
Addison-Wesley, Don Mills Canda, 1979

[9] Pascal/1000 Programmer's Reference Manual

[10] IMAGE Data Base Management System Reference Manual

[11] SPL/3000 Reference Manual

[12] HP 3000 Machine Instruction Set Reference Manual

[13] MPE Intrinsics Reference Manual

[14] Knuth, D. E.
The Art of Computer Programming, Vol. III
Sorting and Searching
Addison-Wesley, Reading, Mass, 1973

[15] Nori et. al.
The Pascal(P) Compiler: Implementation Notes,
Revised Edition
Order from William Waite
Software Engineering Group

Electrical Engineering Department
University of Colorado
Boulder, Colorado 80309

[16] Pascal News
c/o Rick Shaw
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342
Subsrciption rates are $6.00 per year

[17] Pollack, Bary and Greer, David
A Programmer's Introduction to Pascal
Available from Robelle Consulting Ltd.

[18] Wirth, Niklaus; Jensen, Kathleen
Pascal User Manual and Report Second Edition
Springer-Verlag, New York, 1974

[18] Wirth, Niklaus
Pascal-S: A Subset and its Implementation
See order information above

[19] Wirth, Niklaus
Systematic Programming: An Introduction
Prentice-Hall, Englewood Cliffs, New Jersey, 1973

[20] Wirth, Niklaus
Algorithms + Data Structures = Programs
Prentice-Hall, Englewood Cliffs, New Jersey, 1976