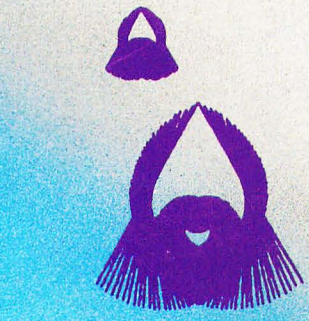
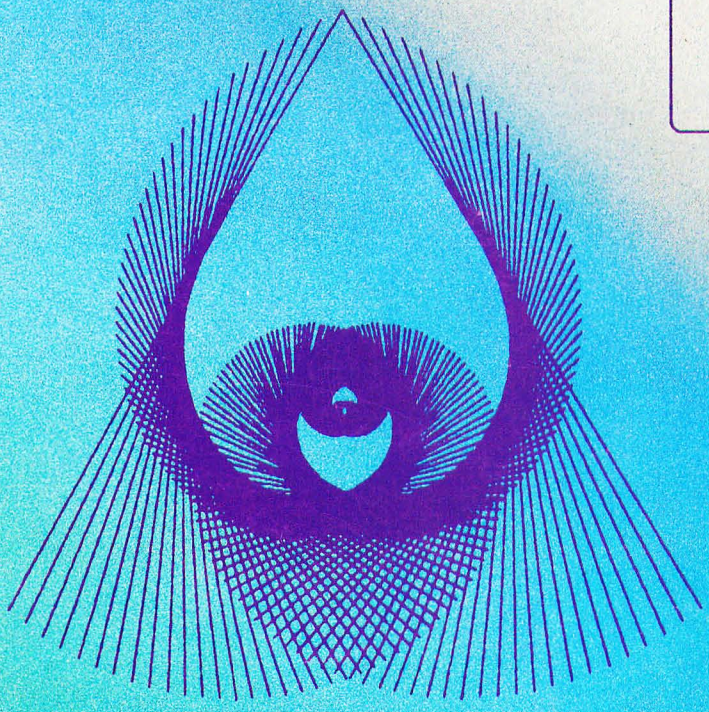
SEPTEMBER 1978 VOLUME 3, Number 9

$2.00 in USA
$2.40 in CANADA

BYTE

the small systems journal





BYTE
BYTE
BYTE
BYTE
BYTE
BYTE
BYTE
BYTE
BYTE
BYTE
BYTE

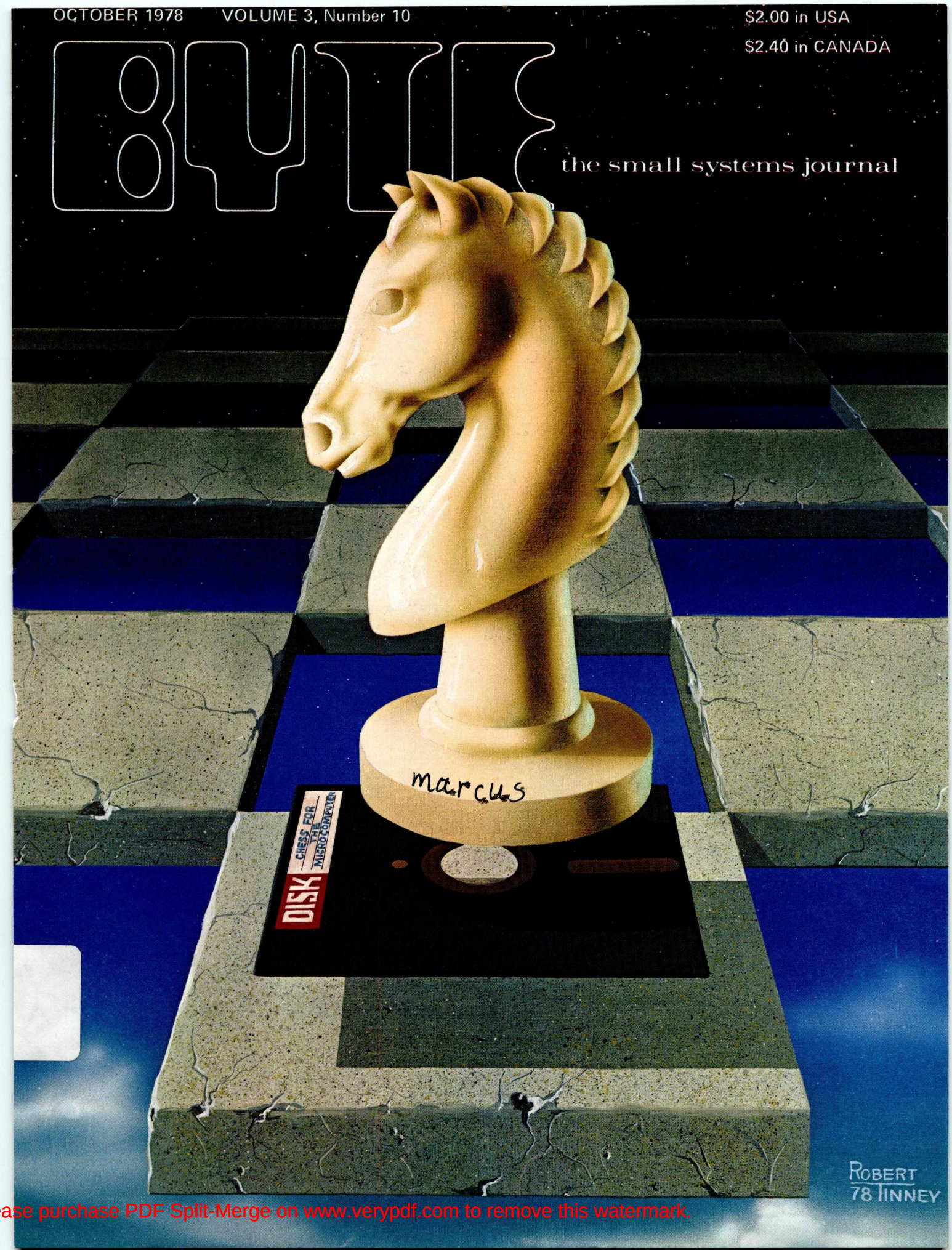
Eduardo Kellerman

OCTOBER 1978 VOLUME 3, Number 10

$2.00 in USA
$2.40 in CANADA

BYTE

the small systems journal

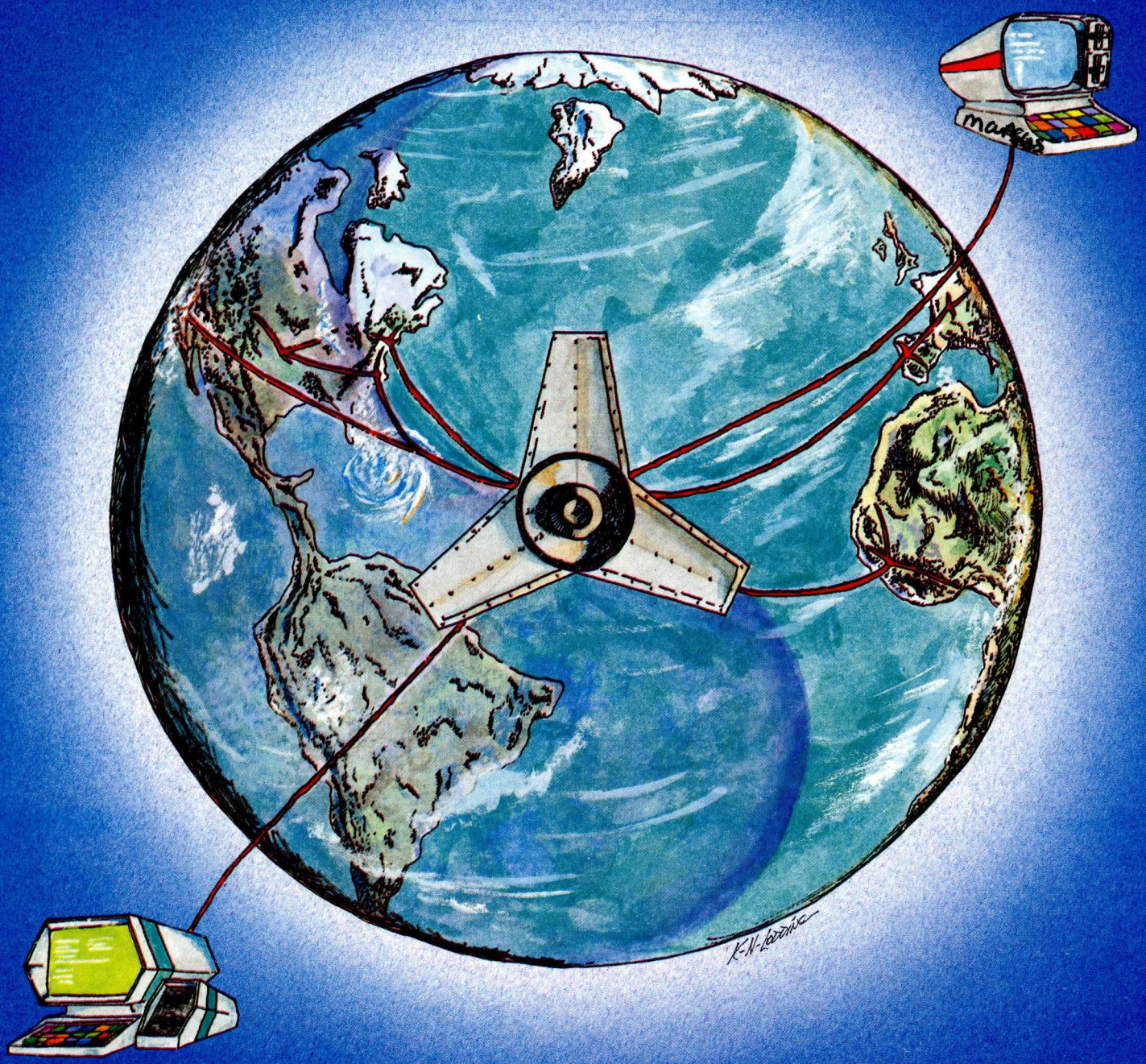


NOVEMBER 1978 VOLUME 3, Number 11

$2.00 in USA
$2.40 in CANADA

BYTE

the small systems journal



A "Tiny" Pascal Compiler

Part 1: The P-Code Interpreter

Kin-Man Chung
124 Scottswood Dr
Urbana IL 61801

Herbert Yuen
POB 2591, Station A
Champaign IL 61820

Roughly speaking, a compiler is a program that translates the statements of a high level language (such as Pascal or FORTRAN) into a semantically equivalent program in some machine recognizable form (such as machine or assembly code). The former is usually referred to as the source program while the latter is called the object program. An interpreter, on the other hand, reads in the source program and starts exection directly, without producing an object program.

There is little doubt that compilers and interpreters are a necessary part of any computer system. The reason most personal computer systems do not have high level language compilers is not that there is no need for them. Compilers, being inherently more complex than interpreters, require more effort to write and more computer memory to run. The main advantage of a compiler over an interpreter is the relative speed. A compiled program typically runs an order of magnitude faster than an equivalent program executed interpretively. In fairness, it must be also pointed out that interpreters are usually easier to use, and more suitable for an interactive environment.

This series of articles is an attempt to describe how a compiler for a subset of Pascal was implemented on an 8080 computer system. It is not our intention to go into details for the reasons for the choice of the language. Pascal is widely recognized as superior to many other languages. For an overview of the language, readers are referred to August 1978 BYTE. The publication, *Pascal: User Manual and Report*, by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1974) should also be consulted as the authoritative source book on the language in its original form.

This is not, of course, the first Pascal compiler ever written for microcomputers. However, instead of waiting for a Pascal compiler to be written for our particular processor, we decided to undertake the project ourselves. In this way, we can add or subtract features from the original Pascal to suit our needs and system capabilities, so that it can be easily integrated with other system software developed so far.

2 Stage Compiler

The compiler is divided into two stages: a p-compiler and a translator. Instead of having the compiler generate machine code directly, it generates code for a hypothetical machine, called the p-machine. These codes, called p-codes, are then converted into the target machine codes by the translator. Dividing the task of a compiler into two stages offers several advantages. The compiler can be written abstractly, without committing oneself to a particular machine and worrying about details of code generation and optimization. Such a compiler is said to be portable, meaning that it can be used on other computer systems with minimal start up effort. It is only at the last stage of code translation from the p-codes to actual machine codes that we have to commit ourselves to a particular machine.

Another advantage this method offers is greater flexibility when writing the compiler. The compiler and the translator can be coded and debugged separately. The flexibility of such a compiler was apparent to us as we started to introduce more and more Pascal features into our original minimal subset. Seldom was it necessary for us to introduce new p-codes other than those originally specified.

There is also one more reason for breaking

About the Authors

Kin-man Chung is currently a graduate student at the University of Illinois at Urbana-Champaign. Presently his equipment includes an Altair 8800 with 44 K bytes of memory, a North Star disk system, a 100 cps impact matrix printer, and a Selectric terminal. His current interest in microcomputers is the development of an interactive Pascal compiler for microcomputers, and a high resolution graphics system capable of animation.

Herbert Yuen received a master of science degree in computer science from the University of Illinois at Urbana-Champaign. He is presently working full-time as a research assistant at the university. His primary interest in microcomputers is software systems development. One of his future plans is to implement a small information retrieval system for a microcomputer.

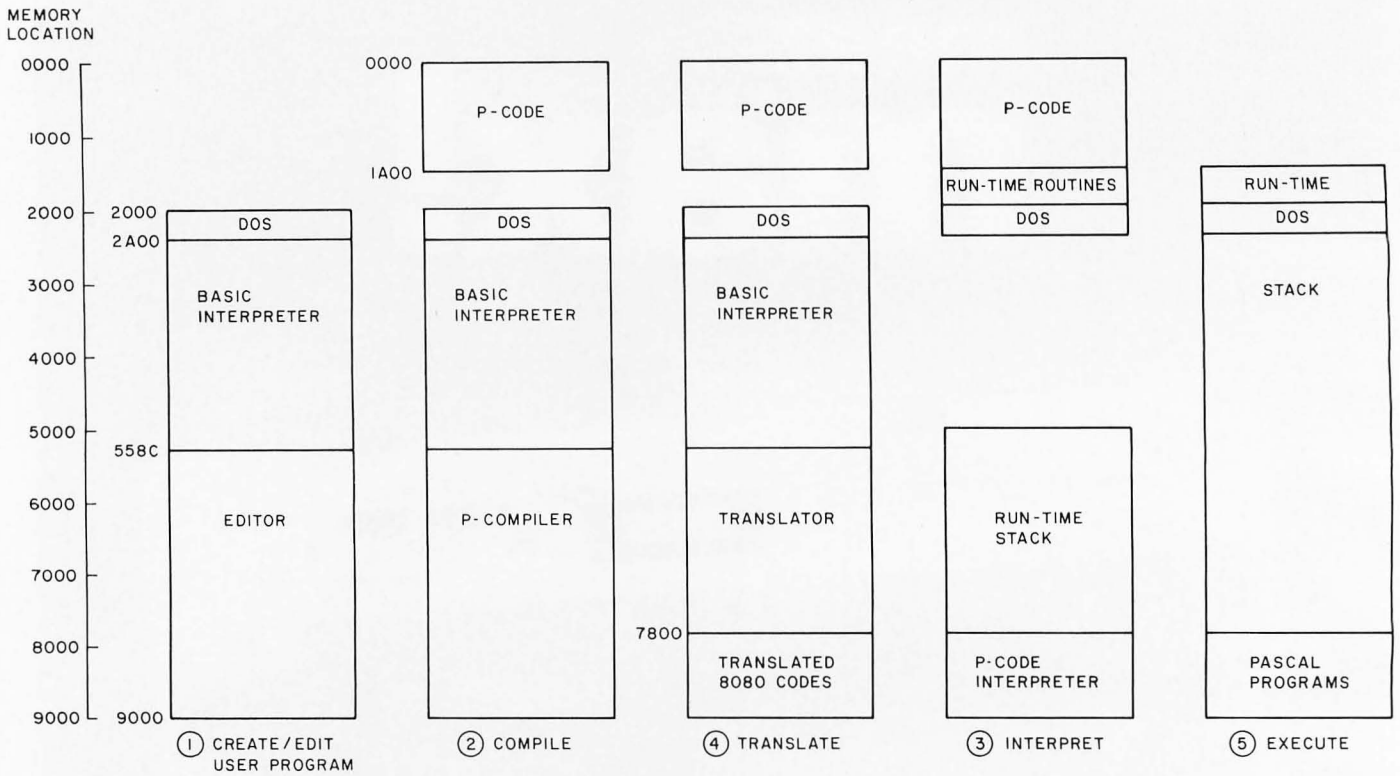


Figure 1: Memory overlay structure of the modules of the compiler. The North Star DOS and BASIC start at hexadecimal 2000 and take up approximately 14 K bytes of memory. The p-compiler is the largest BASIC program of the system; in its compressed form (void of all comments and blanks) it occupies 14 K bytes. It reads Pascal source programs created by the editor from disk files, and generates relocatable p-codes directly in memory. We use hexadecimal 0000 to 19FF for p-codes and find it adequate for Pascal source programs under about 300 lines in length. The smaller translator (9 K bytes) produces 8080 codes directly filled into memory. The origin of the codes can be specified. The run time routines (which total 1 K bytes of memory) are needed only when the translated 8080 codes are being executed. The interpreter is written in Pascal, compiled and translated. The BASIC interpreter is no longer needed when it or any other Pascal program is being run.

the compiler into two stages: most small computers do not have enough memory space to store the complete compiler. After the p-codes are generated, the p-compiler is no longer needed, and can be overlaid with the translator. Therefore the compiler and the translator can share the same memory locations.

Actually we also use two other utility programs: a text editor and a p-code interpreter. The editor is used to prepare the Pascal source programs. The interpreter is used to interpret the p-codes produced by the p-compiler. This provides another alternative for running the Pascal programs. Because it is equipped with various debugging aids, such as setting up breakpoints in p-codes and outputting values for variables, debugging can be easily done. Only after a program is verified to be correct is the translator loaded, and 8080 code produced. This allows easy development of the Pascal programs without sacrifying efficiency at run time. Figure 1 shows the overlay structure for the various modules of the compiler. Figure 2 shows the logical flow during a program development.

In this part of the series on our project, we will describe the general plan. The Pascal subset is defined using syntax diagrams. A description of the p-machine and its codes are also given. We will discuss the p-compiler, translator and runtime routines in the following parts.

Bootstrap Compiler

How does one introduce a new language into a computer system with limited computer resources? By computer resources we mean not only the computer hardware like memory and peripherals, but also software tools. We have learned from experience not to attempt programs with the complexity of a compiler in machine or assembly language. This left us with BASIC. Although it is not the most desirable language to write a compiler with, it turned out to be adequate. Some careful thought is needed, of course, to handle recursive subroutine calls from BASIC, a feature central to our compiler writing.

The alternative to BASIC is to go to a commercial computer and write the whole or part of the compiler in an appropriate

language. The finished product (or part of it) can then be transferred to the smaller computer. This is, however, a luxury most of us cannot afford.

Of course, the compiler written in BASIC

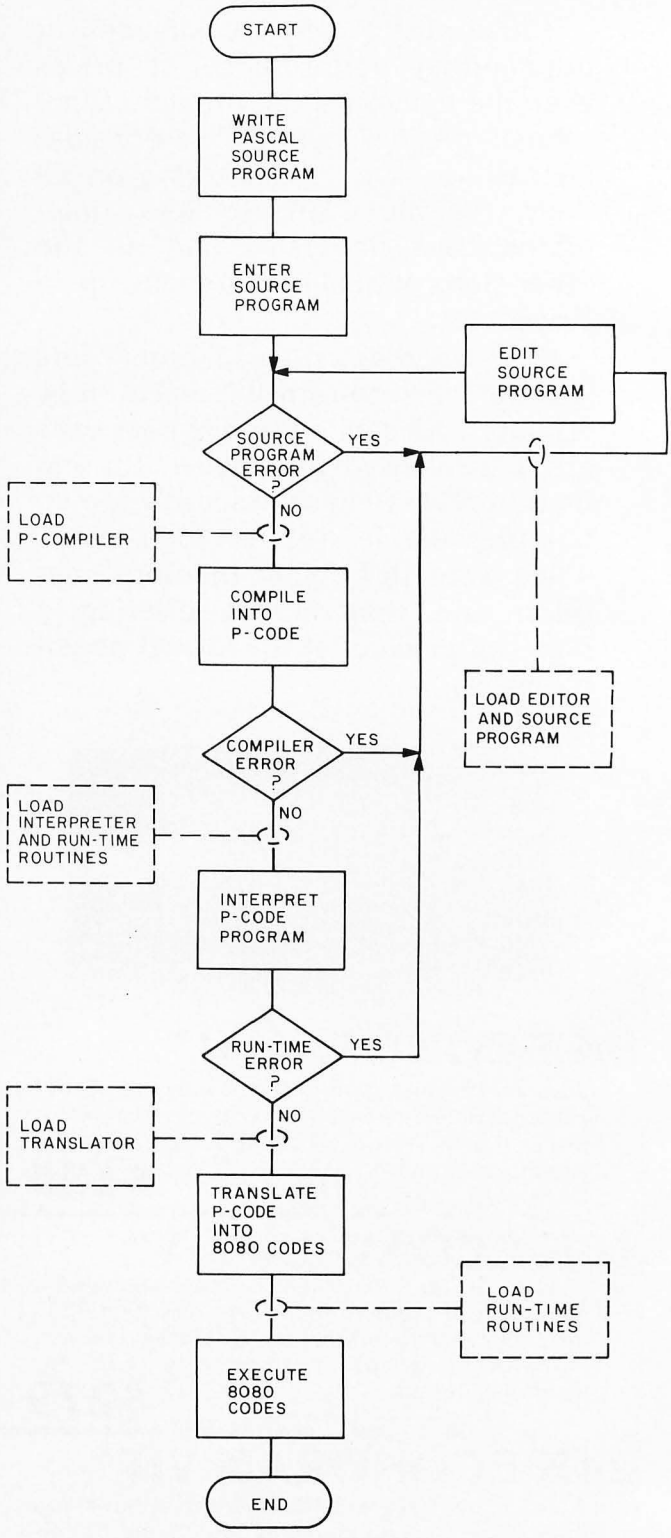


Figure 2: Flowchart showing development of a Pascal program.

would be very inefficient and slow. But this actually would not matter, since it would only be used as a *bootstrap* compiler. The concept of bootstrapping should be familiar to most personal computer owners. We usually use it when initially starting up our computers. After turning on the power, a bootstrap loader is first loaded into the computer (either manually or through the use of read only memory). This bootstrap loader is then used to load the loader, which in turn loads the monitor into memory. The bootstrap loader is a smaller version of the loader; it is just big enough to load the main loader and not adequate to be a general purpose loader.

The same idea can be applied to compiler writing. A compiler for a small subset of a language is first written. This subset should be big enough so that a compiler for a bigger subset of the same language can be written in it. The larger compiler is then written and compiled, using the first compiler. Next, a compiler for a still bigger subset of the same language can then be written and compiled, using the second compiler, and so on until a compiler for the complete language is produced. In actual practice, no more than three stages are used. It does not matter if the first compiler is very inefficient. The idea is to get a working, albeit primitive and inefficient, compiler with minimum starting effort.

Pascal Subset Syntax

The syntax of Pascal can be described precisely by using a notation usually called Backus-Naur form (BNF). This is a collection of rules for the grammar of the language. Instead of dealing with Backus-Naur form directly, we use an equivalent but more understandable notation: the syntax diagrams. Figure 3 describes the syntax of the Pascal subset we are interested in.

In the syntax diagram, the square boxes are called nonterminal symbols, while the ovals are called terminal symbols. Terminal symbols are the basic building units of the language and require no further expansion. In our case, the names that represent the terminals are also their textual representations in the language. The nonterminal symbols in the syntax diagrams can be expanded using rules specified in another syntax diagram, and there is a syntax diagram for each nonterminal symbol in the syntax diagram. A branch in the diagram represents options allowable by the grammar. When all nonterminal symbols are eliminated by expansion in this fashion, we would have a valid program. We start off a compilation with the nonterminal program. Looking at

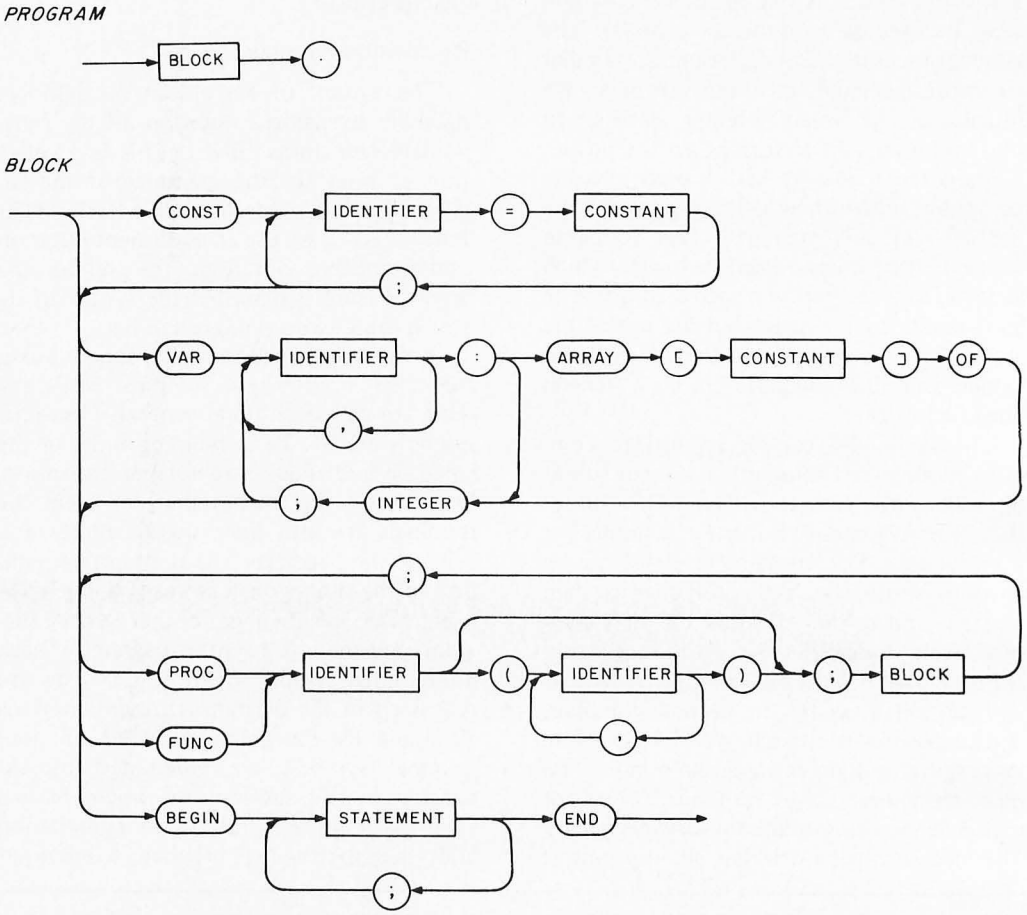


Figure 3: Syntax diagrams of the Pascal subset. For the syntax diagrams of the full Pascal set refer to the book by Kathleen Jensen and Niklaus Wirth, entitled Pascal: User Manual and Report. *These diagrams totally define the subset of the language that we are using.*

the syntax diagram we see that a program is a block followed by a period (.). Looking at the syntax diagram for block, we notice that it can have an optional declaration part followed by the main body which begins with the string **begin**, followed by any number of the nonterminal symbols, statement, separated by semicolons (;), and then the string **end**. The statement block can be further expanded by the syntax diagram for statement, and so on.

The reason we go through the details here is because it is important to precisely describe the features we want to include in our language before starting to write the compiler. It is the first step towards writing the compiler. These syntax diagrams will later become flowcharts for the syntax analyzer of the compiler.

Readers familiar with Pascal will no doubt notice several important features missing from our subset. There is no GOTO statement. The only data type we have is integer and integer array of one dimension. Also missing from the subset is the structured data type, pointer type, user defined type, and file type. A less obvious omission is passing the parameter of a procedure by address; the parameters are passed by value only. Aside from the fact that these features are difficult to implement, they are not indispensable in our bootstrap process. Of course, features like user defined type and structured type are some of the unique features of Pascal, and should not be omitted in the long run. But we feel that they can be added later.

We have also included some trivial but nevertheless useful enhancements to the language, which we hope do not deviate from the standard too much. One is the addition of the optional clause **else** to the **case** statement which provides an exit path if the value of the variable does not fall into any of the **case** labels. Another is the inclusion of format controls in the read and write statements. Following an expression in a *write* statement, a pound sign, #, indicates numeric form and a percent sign, %, indicates hexadecimal format. If there is no

format control, a character whose ASCII code equals the expression is output. Also a hexadecimal constant is prefixed by %. This allows processing of hexadecimal numbers without conversion by the user.

To allow interfacing Pascal programs with assembly programs, a facility is provided to read or write a byte from or to absolute memory locations. The array *mem* is a reserved array name that is used to do this.

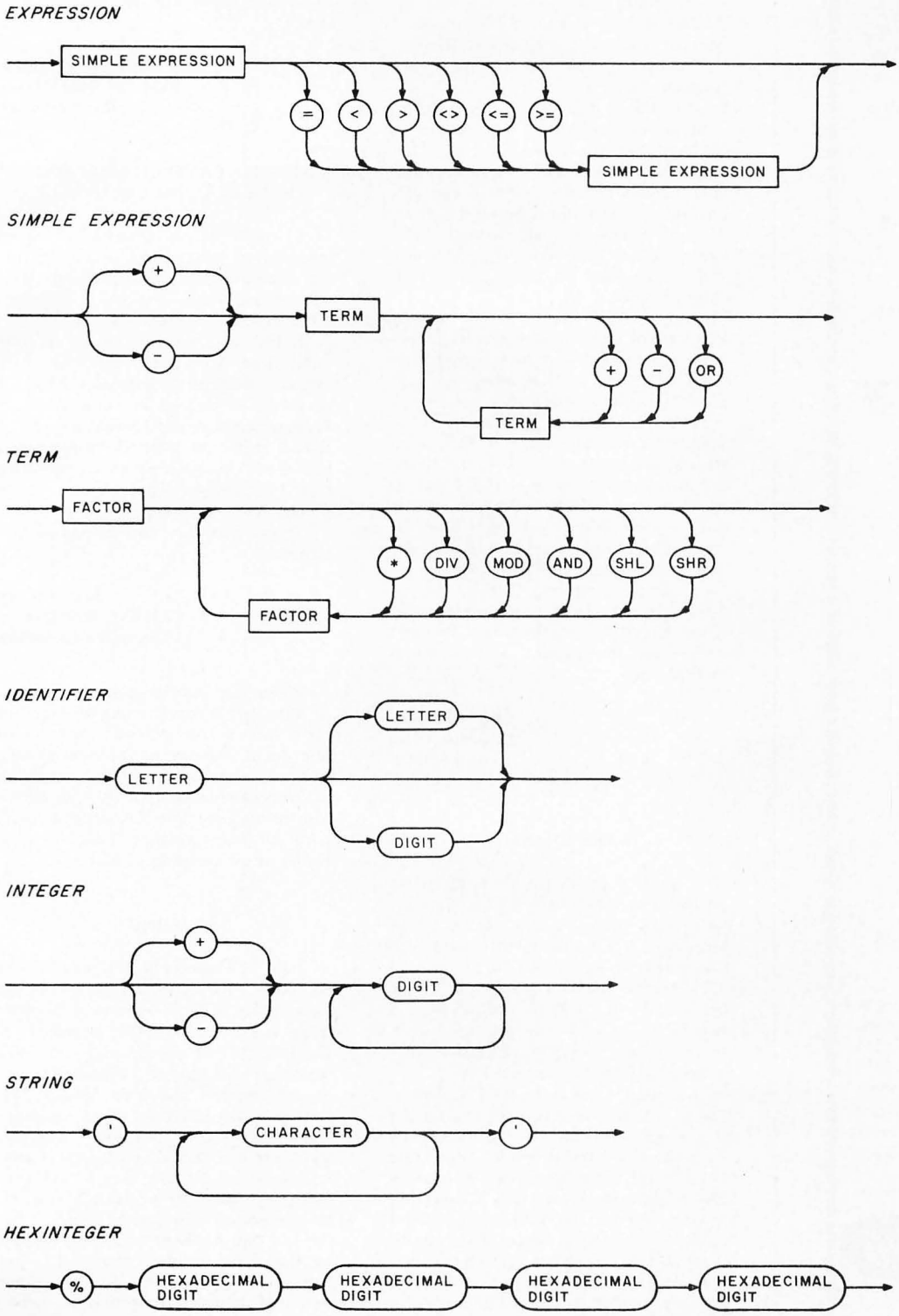


Figure 3, continued: Elementary constructs for Pascal subset. Hexinteger is usually not defined in Pascal but is used here so that actual memory locations can be easily manipulated.

Continued on page 149

Continued from page 65

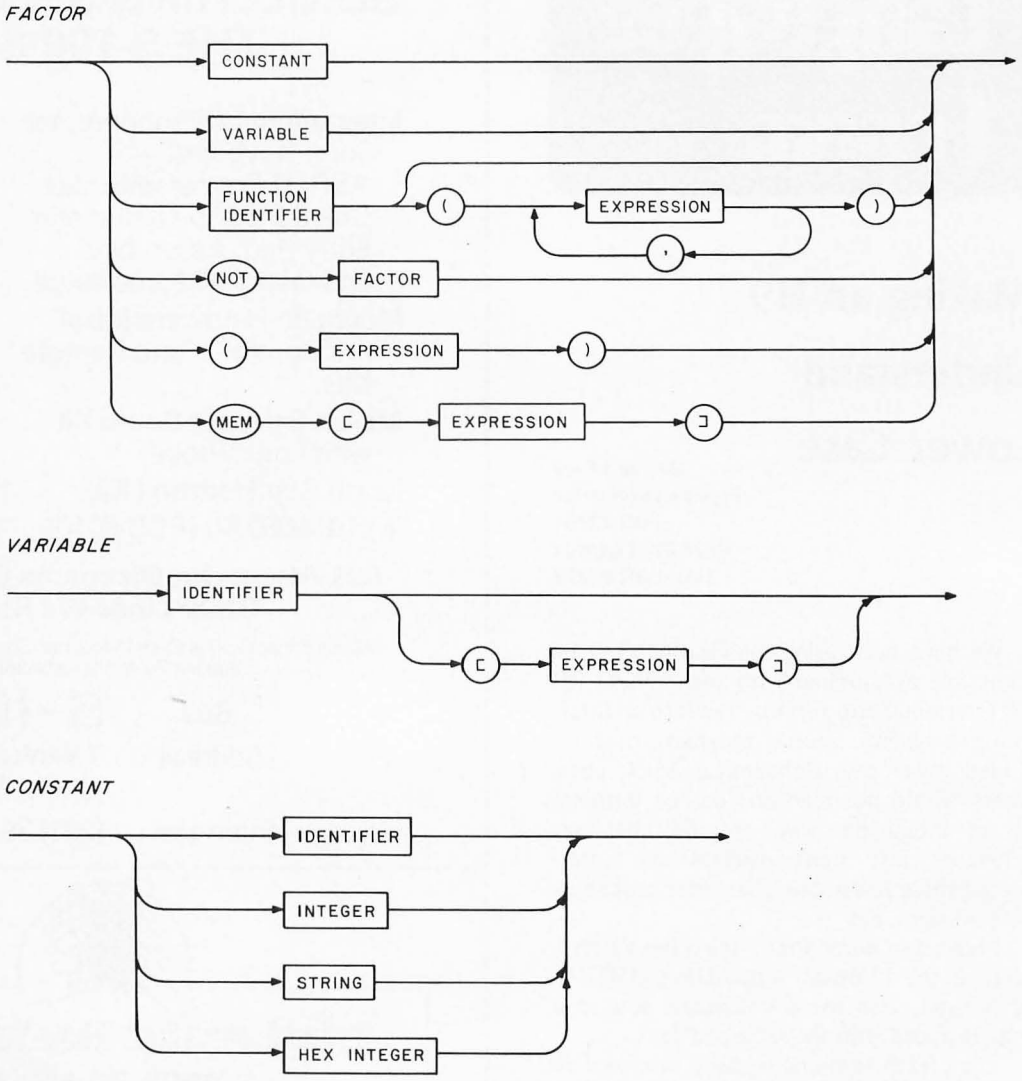


Figure 3, continued: Notice that some of the diagrams, for example FACTOR, contain themselves in their own definitions. This is known as a recursive definition.

For instance:

mem [i]:=mem[j];

reads the byte from the memory location *j* and writes it back to memory location *i*. Machine language subroutines can be called from Pascal programs. The statement:

Call (i);

can be used to make a call to memory address *i*.

The P-Machine

The p-machine is a stack oriented machine consisting of four registers and two memory storage areas. Memory is separated into program storage and data storage areas. The program storage area contains the program codes (p-codes), and remains unchanged during program execution. The data storage area contains the values of variables. It is also used to store temporary values during arithmetical and logical operations.

Though the variables can be fetched and stored in a random fashion, the data storage area operates as a stack with respect to arithmetical and logical operations and runtime storage allocation. Arithmetical and logical operations are done on the top elements of the stack, and the results of the operations are pushed back on the stack. In this respect, one might call it a zero address machine, since operations (except store and load instructions, which must specify an address) are done without reference to any address. Later we will discuss the use of the stack during runtime storage allocation.

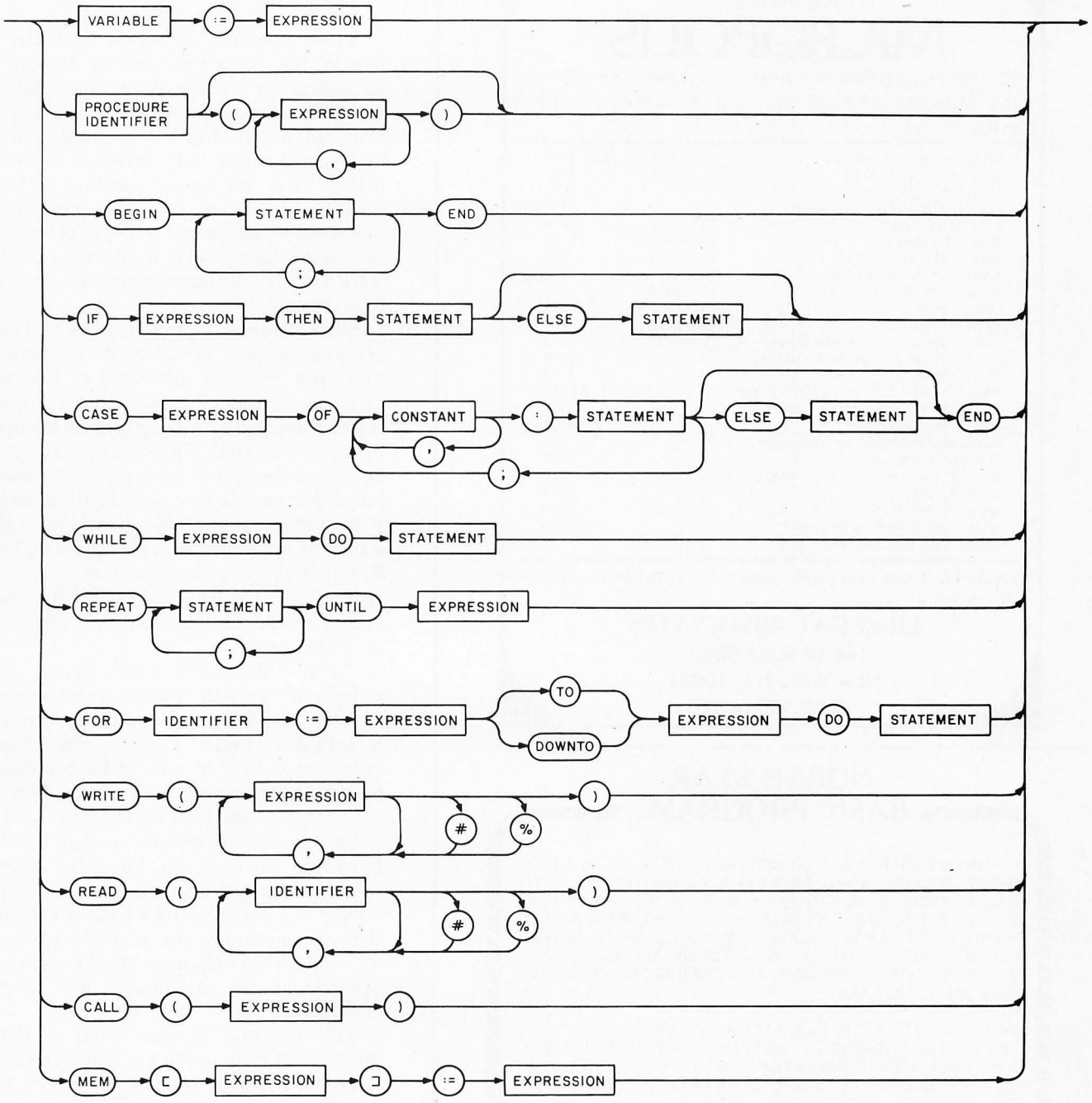


Figure 3, continued.

The four registers in the p-machine are the program counter, P, which points to the next executable instruction in the program storage; the instruction register, I, which contains the current execution instruction; the stack pointer, T, which points to the top of the stack, and the base address register, B, which contains the current base address. The functions of the first three registers should be quite clear from the above discussion. The function of register B will become clear after we discuss storage allocation.

Each variable in a Pascal procedure has a scope and lifetime. The scope of a variable is the range within which it can be referenced. The scope of a Pascal variable is simply the procedure block to which it belongs. The lifetime of a variable is from the time storage is allocated for it to the time storage is disallocated. In Pascal, this is the time the procedure defining the variable is activated to the time a return is executed by the procedure. This is different from the way variables are treated in BASIC, where the scope of a variable is the entire program and its lifetime the entire execution time.

Since procedure activation is strictly a first in, last out process, the use of stack is an appropriate strategy. When a procedure is activated, storage for its local variables is allocated on the top of the stack, and is disallocated when the procedure is terminated. Thus the stack contains all the variables of the currently active procedures. The variables of the last activated procedure are on the top of the stack, those of the second to last activated procedure next to it, and so on.

Since storage allocation is not static, addresses cannot be assigned at compile time, but must be calculated at runtime. The base register, B, always points to the starting location of the segment of the data block in the stack. The addresses generated by the compiler are not absolute addresses, but displacements from some base addresses. If the variable is local, then its address is the displacement from the current base register B; but if the variable is from an outer procedure, then the base address for that procedure should be calculated, and added to the displacement.

To do this, and to ensure proper procedure or function linkage, extra storage is allocated on the stack when a procedure is activated. Figure 4 shows the various quantities present in each of the procedure blocks. The function return value is used only for function calls, and storage is allocated for any parameters needed by the procedure or function. The base address contains the value of the current base register B, and the return address contains the program return address at the place of the call. The functions of the dynamic linkage and the static linkage need further explanation.

The dynamic linkage forms a chain that reflects the procedure activation history. It points back to the base address of the procedure that was activated immediately before this one. For instance, if procedure A calls procedure B, which calls procedure C, then the dynamic link chain points from C to B, and then to A. It is used to ensure that the program returns to its previous state when exiting a procedure. In particular, the base register B must be loaded with the correct base address of the calling procedure. This would be easy to do if we follow a step through the dynamic link chain.

The static link, on the other hand, reflects the static hierarchical structure of the procedures. Each active procedure has a link that points to the procedure (also active) that immediately contains it. The static links actually form a tree, with the

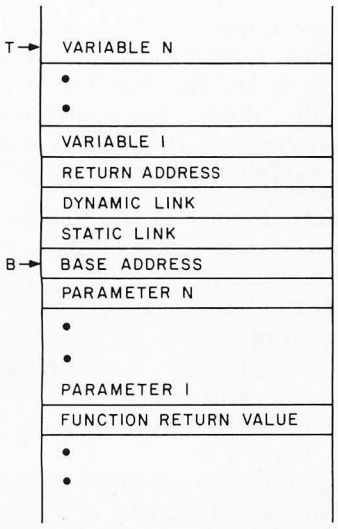


Figure 4: A typical activation record for a function. For a procedure, the function return value is omitted. Note that the procedure and function parameters, as well as the function return value, are below the base register B, and thus would have negative displacements.

main program block as the root. These links, which in general are different from the dynamic links, are used to let programs have access to the correct base address of the variables in an outer procedure, since at compiler time, only the static relationship among the procedures are known. The compiler therefore generates the pair (static level difference, relative displacement from the base address) as addresses for variables. The calculation of the addresses from these pairs would presumably slow down the process, but it is a small price to pay for nice features like recursive procedure calls.

Op Code (Hexadecimal)	Mnemonic		Operation
00	LIT	0,n	load literal constant
01	OPR	0,n	arithmetic or logical operation
02	LOD	v,d	load variable
12	LODX	v,d	load indexed variable
03	STO	v,d	store variable
13	STOX	v,d	store indexed variable
04	CAL	v,a	call procedure or function
05	INT	0,n	increment stack pointer
06	JMP	0,a	jump unconditional
07	JPC	c,a	jump conditional
08	CSP	0,n	call standard procedure

Table 1: Basic p-codes. The v in call, load and store instructions is the difference in static level between the current procedure and the one being called or the one which contains the variable from the base address. An address in a p-code program is shown by a. The condition code, c, can either be 0 or 1.

The P-Codes

The p-machine has only 11 basic instructions, which are listed in table 1. For the sake of simplicity and easy handling in this version of the implementation, all instructions are four bytes long. The contents of the four bytes are as follows:

byte 1: op — the operation code.
byte 2: can be (i) v — static level difference.
or (ii) c — condition code in a jump instruction.
or (iii) 255 — denotes absolute addressing.
or (iv) not used for some instructions.
bytes 3,4: can be (i) d — displacement from the base address.
or (ii) n — numeric constant.
or (iii) a — address in the p-code program.

The OPR (arithmetic and logical operations) and CSP (call standard procedure) are further subdivided into more instructions. The complete set of instruction mnemonics and operations is listed in table 2. The LODX and STOX instructions are used to load and store array elements with the value of the array subscript on top of the stack. The call standard procedure (CSP) instruction is primarily used for input and output (IO) operations. Besides the basic function of inputting and outputting single characters, additional procedures have been implemented to relieve the user from writing IO conversion routines in Pascal for numeric and hexadecimal numbers. In the future, more procedures can be added to handle the input and output of other data types such as floating point numbers and file records for tape or disk. Meanwhile these seven instructions are sufficient for convenient use in writing the bootstrap compiler and its related software.

Readers are urged to read the p-code interpreter listing which simulates the operations of the p-machine. The program statements are straightforward and self-explanatory. Familiarity with the p-machine instruction set is essential in understanding the code generation part of the p-compiler.

The P-Code Interpreter

Since the p-machine is a hypothetical computer, there has to be some method of executing the p-codes generated by the compiler. There are two simple solutions to this problem. One is to write an interpreter which can decode and execute the p-codes. The other solution is to write a trans-

lator which can decode the p-codes and output equivalent executable machine codes for an existing computer. Both methods have been used in our compiler system. The first method, although it runs slower, is good for developing programs because many debugging facilities can be implemented in the interpreter. The second method is good for production programs which may need faster execution speed. A p-code to 8080 machine code translator will be described in part 3 of this series.

The p-code interpreter is made up of two major modules:

- Main program.
- Procedure which simulates the p-machine.

Every call to the simulator will execute one p-machine instruction. Each p-machine instruction cycle can be divided into four stages:

- Fetch a p-code from memory.
- Increment the program counter.
- Decode the instruction.
- Execute the instruction.

Several global variables are used to hold the values of the p-machine registers such as program counter, stack pointer, current instruction, etc. A one-dimensional array represents the data stack. Functional operations of the various p-machine instructions are coded directly from the instruction set defined in table 2. The main program simply initializes the program counter to zero and then calls the simulator repeatedly to simulate machine execution. This sounds simple but not useful, because the user has no control of the program during execution until it terminates.

In order to enable user control of an executing p-code program, the main program must accept commands from the user which instruct it to call the simulator a specified

G: *go* — Set program counter to zero; initialize other counters; start execution.
S: *single-step* — Execute one p-code; display the mnemonics of the next p-code pointed by the updated program counter.
R: *run/restart* — Start execution from current program counter until the program ends or a breakpoint is reached. This command is used to continue execution at a breakpoint.
B: *set breakpoint* — A p-code address is entered as a breakpoint after the interpreter prompts with a ?. Up to five breakpoints may be set.
C: *clear* — All breakpoints previously set are cleared.
Y: *display breakpoint* — Display the breakpoints already set.
X: *examine status* — Display the values of: current program counter, base address, stack pointer, the top two elements of the stack.
K: *stack content* — A value is entered as the stack pointer after the interpreter prompts with a ?. It will then display the values of six stack elements starting from this stack pointer.
T: *trace* — Display the address and mnemonics of the 16 p-codes last executed. This command is usually applied at a breakpoint. It is used for tracing the logic flow of the program.
E: *examine program* — A p-code address is entered as a display pointer (DP) after the interpreter prompts with a ?. It will then display the mnemonics of the p-code at this address. This command and the U and N commands are used for examining the p-codes anywhere in the program without altering the current program counter.
U: *up* — Decrement the display pointer by one and display the mnemonics of the p-code pointed by it.
N: *next* — Increment the display pointer by one and display the mnemonics of the p-code pointed by it.
Q: *quit* — Terminate the interpreter program and return to operating system.

Table 3: Interpreter commands. All commands for the p-code interpreter are single characters. A command is entered after the interpreter prompts the user with a > on the video display. Additional information is needed for some commands such as breakpoint and stack addresses. On entry to the interpreter it will ask for the starting memory address of p-codes and initialize the program counter to zero. On exit it will display the number of p-codes executed.

Mnemonic	Description
LIT 0, n	load literal constant
OPR 0, 0	procedure return
OPR 0, 1	negate (sp)
OPR 0, 2	add (sp) to (sp−1)
OPR 0, 3	subtract (sp) from (sp−1)
OPR 0, 4	multiply (sp−1) by (sp)
OPR 0, 5	divide (sp−1) by (sp)
OPR 0, 6	low order bit of (sp)
OPR 0, 7	(sp−1) modulo (sp)
OPR 0, 8	test for (sp−1)=(sp)
OPR 0, 9	test for (sp−1)<>(sp)
OPR 0,10	test for (sp−1)<(sp)
OPR 0,11	test for (sp−1)>=(sp)
OPR 0,12	test for (sp−1)>(sp)
OPR 0,13	test for (sp−1)<=(sp)
OPR 0,14	logical (sp−1) OR (sp)
OPR 0,15	logical (sp−1) and (sp)
OPR 0,16	logical NOT of (sp)
OPR 0,17	shift left (sp) logical
OPR 0,18	shift right (sp) logical
OPR 0,19	increment (sp) by 1

Mnemonic	Description
OPR 0,20	decrement (sp) by 1
OPR 0,21	copy (sp) to (sp+1)
LOD v,d	load a word
LOD 255,0	load a byte from absolute address (sp)
LODX v,d	load a word with index address (sp)
STO v,d	store a word
STO 255,0	store a byte to absolute address (sp−1)
STOX v,d	store a word with index address (sp)
CAL v,a	procedure call
CAL 255,0	call procedure at absolute address (sp)
INT 0,n	increment sp by n
JMP 0,a	jump to location a
JPC 0,a	jump to location a if low order bit (sp)=0
JPC 1,a	jump to location a if low order bit (sp)=1
CSP 0,0	input 1 character
CSP 0,1	output 1 character
CSP 0,2	input an integer
CSP 0,3	output an integer
CSP 0,4	input a hexadecimal number
CSP 0,5	output a hexadecimal number
CSP 0.8	output a string

Table 2: The p-machine instruction set. The stack pointer, sp, points to the top element of the stack. The content of the stack element is represented by (sp). The operands of the OPR instructions are replaced by their results on the stack. The result of the six relational operations is 1 if the test is true and 0 if false. With the exception of single operand OPR instructions, all instructions adjust the stack pointer, sp, after execution.

North Star BASIC

A brief summary of North Star BASIC (version 6, release 3) is given for readers not familiar with its particular features.

Variable names are one or two characters long: an alphabetical character followed optionally by a decimal digit. There are four types of variables: numeric, string, array of numeric, and function. The string variables are names postfixed by a dollar sign $, while function names are prefixed by FN. Functions (and the parameters) are defined by the declaration DEF, and ended by FNEND (for multiline function). The parameters in the function definition are local to the function, and would not affect variables in the calling program.

Strings cannot be dimensioned. The DIM declarations for strings declare the maximum length of the string variables, not their dimensions. The notation A$ (3, 5) denotes the substring of A$ from position 3 to 5. Thus if A$=ABCDEFG, A$ (3, 5) is the string CDE. This substring expression can be used both on the left or righthand side of an assignment statement.

Multiple statement lines are allowed. Statements within a line are separated by either colons, :, or back slashes, \.

Absolute memory locations can be accessed from BASIC programs. The function EXAM(I) returns the content of memory at address I; and the instruction FILL I, J writes a value of J into memory address I.

Another feature of North Star BASIC is its ability to read from or write to disk files. The statement OPEN #0, "FNAME" assigns disk file "FNAME" to file unit #0. A subsequent READ #0,A$ reads A$ from the disk file, and a WRITE #0,A$ writes A$ to the disk file. A built-in function TYP can be used to check the type of data to be read. It has a value of 0 when the end of file is reached.

number of times or to display register and stack contents. This is the simple idea of a debugging interpreter. The debugging aids commonly known include single step execution, set and reset of breakpoints, and display of register and stack contents. A number of these debugging facilities have been incorporated in the p-code interpreter. Table 3 shows the 13 interpreter commands and their functions. Note that the trace command is particularly useful in analyzing mysterious logic flow of a program, such as discovering the path along which a breakpoint is reached. This command is more convenient to use and much faster than single step execution. The limits on the number of breakpoints and the number of instructions traced can be changed easily in the program.

The first version of the p-code interpreter was written in BASIC. While developing the p-compiler, different constructs of Pascal statements were tested one at a time using the interpreter to verify the correctness of the p-codes generated. After the compiler was debugged, the interpreter was rewritten in Pascal. The program logic is very similar to the BASIC version. Since the program

Listing 1: Pascal source code for the p-code interpreter as output by the authors' system. This version implements all of the commands in table 3.

```
P-CODES STARTS AT 0000
WANT CODE PRINTED?N
   0 ?$P.INTS
   0 ( P-CODE INTERPRETER.  HY.1  3/31/78   BY  H YUEN )
   0     ( LAST MOD 4/12/78 )
   0 CONST U=15;BPLIM=5;SIZE=500;SIZE1=480;
   1 VAR Z,P,B,T,BP,P0,TP,CMD,I,J,K,STOP:INTEGER;
   1      S:ARRAY[SIZE] OF INTEGER;
   1      TRACE:ARRAY[U] OF INTEGER;
   1      MN:ARRAY[26] OF INTEGER;
   1      BREAK:ARRAY[BPLIM] OF INTEGER;
   1
   1    ( IMPORTANT GLOBAL VARIABLES:
   1      P:PROGRAM COUNTER       B:BASE POINTER
   1      T:STACK POINTER        BP:BREAK POINT INDEX
   1     TP:TRACE STACK PTR       K:INSTRUCTION COUNTER
   1      S:DATA STACK            Z:STARTING ADDR OF P-CODE )
   1
   1 FUNC BASE(LEV);
   1   VAR B1:INTEGER;
   2   BEGIN B1:=B;
   5    WHILE LEV>0 DO BEGIN
   9     B1:=S[B1];LEV:=LEV-1 END;
  17    BASE:=B1
  18   END (BASE);
  20
  20 PROC INIT;
  20   VAR I:INTEGER;
  21   BEGIN T:=0;B:=1;P:=0;STOP:=0;
  30    S[1]:=0;S[2]:=0;S[3]:=-1;
  40    P0:=0;TP:=U;K:=0;
  46    FOR I:=0 TO U DO TRACE[I]:=-1
  55   END (INIT);
  63
  63 PROC CRLF;
  63   BEGIN WRITE(13,10) END;
  70
  70 PROC EXEC;
  70   VAR X,A,L,F,IDX:INTEGER;
  71   BEGIN X:=P SHL 2 + Z;
  78    A:=MEM[X+3] SHL 8 +MEM[X+2];
  90    TP:=TP+1;IF TP>U THEN TP:=0;
 100    TRACE[TP]:=P;
 103    P:=P+1;P0:=P;K:=K+1;
 113    F:=MEM[X];
 116    IF F<=8 THEN IDX:=0
 121     ELSE BEGIN IDX:=1;F:=F-16 END;
 129    CASE F OF
 130 0:BEGIN T:=T+1;S[T]:=A END;
 142 1:  CASE A OF
 147     0 :BEGIN (RETURN)
 151         T:=B-1;B:=S[T+2];P:=S[T+3] END;
 166     1 :S[T]:=-S[T];
 176     2 :BEGIN T:=T-1;S[T]:=S[T]+S[T+1] END;
 194     3 :BEGIN T:=T-1;S[T]:=S[T]-S[T+1] END;
 212     4 :BEGIN T:=T-1;S[T]:=S[T]*S[T+1] END;
 230     5 :BEGIN T:=T-1;S[T]:=S[T] DIV S[T+1] END;
 248     6 :S[T]:=S[T] AND 1; (TEST FOR ODD)
 259     7 :BEGIN T:=T-1;S[T]:=S[T] MOD S[T+1] END;
 277     8 :BEGIN T:=T-1;S[T]:=S[T]=S[T+1] END;
 295     9 :BEGIN T:=T-1;S[T]:=S[T]<>S[T+1] END;
 313    10:BEGIN T:=T-1;S[T]:=S[T]<S[T+1] END;
 331    11:BEGIN T:=T-1;S[T]:=S[T]>=S[T+1] END;
 349    12:BEGIN T:=T-1;S[T]:=S[T]>S[T+1] END;
 367    13:BEGIN T:=T-1;S[T]:=S[T]<=S[T+1] END;
 385    14:BEGIN T:=T-1;S[T]:=S[T] OR S[T+1] END;
 403    15:BEGIN T:=T-1;S[T]:=S[T] AND S[T+1] END;
 421    16:S[T]:=NOT S[T];
 431    17:BEGIN T:=T-1;S[T]:=S[T] SHL S[T+1] END;
 449    18:BEGIN T:=T-1;S[T]:=S[T] SHR S[T+1] END;
 467    19:S[T]:=S[T]+1;
 478    20:S[T]:=S[T]-1;
 489    21:BEGIN (COPY)
 493         T:=T+1;S[T]:=S[T-1] END
 503      ELSE BEGIN WRITE(' ILLEGAL OPR');CRLF;STOP:=1 END
 521   END (CASE OF A);
 523 2:BEGIN (LOAD)
 527    L:=MEM[X+1];
 532    IF L=255 THEN S[T]:=MEM[S[T]]
 539    ELSE BEGIN IF IDX THEN A:=A+S[T];
 549          T:=T+1-IDX;S[T]:=S[BASE(L)+A] END
 564   END;
 565 3:BEGIN (STORE)
 569    L:=MEM[X+1];
 574    IF L=255 THEN BEGIN
 578     MEM[S[T-1]]:=S[T];T:=T-2 END
 589    ELSE BEGIN
 590     IF IDX THEN A:=S[T-1]+A;
 599     S[BASE(L)+A]:=S[T];T:=T-1-IDX END
 614   END;
 615 4:BEGIN (CALL)
 619    L:=MEM[X+1];
 624    IF L=255 THEN BEGIN CALL(S[T]);T:=T-1 END
 635    ELSE BEGIN
 636     S[T+1]:=BASE(L);S[T+2]:=B;
 649     S[T+3]:=P;B:=T+1;P:=A END
 660   END;
 661 5:IF T>(SIZE1-A) THEN BEGIN
 671    WRITE(' STACK OVFL');CRLF;STOP:=1 END
 687   ELSE T:=T+A;
 693 6:P:=A; (JMP)
 700 7:BEGIN IF S[T]=MEM[X+1] THEN P:=A;  (JPC)
```

```
714     T:=T-1 END;
719  8:CASE A OF (CSP)
724    0:BEGIN T:=T+1;READ(S[T]) END;   (IN CHAR)
736    1:BEGIN WRITE(S[T]);T:=T-1 END;  (OUT CHAR)
748    2:BEGIN T:=T+1;READ(S[T]#) END;  (IN NUMBER)
760    3:BEGIN WRITE(S[T]#);T:=T-1 END;(OUT NUMBER)
772    4:BEGIN T:=T+1;READ(S[T]%) END;  (IN HEX)
784    5:BEGIN WRITE(S[T]%);T:=T-1 END;(OUT HEX)
796    8:BEGIN (OUT STRING)
800        FOR IDX:=T-S[T] TO T-1 DO WRITE(S[IDX]);
820        T:=T-S[T]-1 END
827    ELSE BEGIN WRITE(' ILLEGAL CSP');CRLF;STOP:=1 END
845    END (CASE OF A)
846  ELSE BEGIN WRITE(' ILLEGAL OPCODE');CRLF;STOP:=1 END
867   END (CASE OF F)
868  END (EXEC);
869
869  PROC CODE(PC); (PRINT CODE)
869   VAR X,N,IDX:INTEGER;
870   BEGIN X:=PC SHL 2 +Z;N:=MEM[X]#3;
882    IF N<=24 THEN IDX:=' '
887    ELSE BEGIN N:=N-48;IDX:='X' END;
895    WRITE('  ',PC#,'  ',MN[N],MN[N+1],MN[N+2],IDX,'
924     MEM[X+1]#,',',MEM[X+3] SHL 8 +MEM[X+2]#);CRLF
944   END (CODE);
945
945  PROC CKBP; (CHECK BREAK POINT)
945   VAR I:INTEGER;
946   BEGIN IF P<0 THEN STOP:=1
952    ELSE BEGIN
954    FOR I:=1 TO BP DO
961     IF BREAK[I]=P THEN BEGIN
```

```
966       WRITE(' BREAK:');CODE(P);
978        STOP:=1 END END
985   END (CKBP);
986
986  BEGIN (MAIN)
986   FOR I:=0 TO 26 DO
994    MN[I]:=MEM[I+%1E80]; (MNEMONICS ARE IN MEMORY)
1005  WRITE('ADDR?');READ(Z%);CRLF;
1015  INIT; CODE(P);BP:=0;
1021  REPEAT WRITE('>');READ(CMD);
1025   CASE CMD OF
1026 'R':BEGIN STOP:=0;REPEAT EXEC;CKBP UNTIL STOP END;
1037 'S':BEGIN EXEC; CODE(P) END;
1046 'X':BEGIN
1050       WRITE('  P=',P#,' B=',B#,' T=',T#,
1072            ' S[T]=',S[T]#,' S[T-1]=',S[T-1]#);CRLF
1099      END;
1100 'G':BEGIN INIT;REPEAT EXEC;CKBP UNTIL STOP END;
1110 'T':BEGIN WRITE(' *TRACE#');CRLF;
1125       FOR I:=0 TO U DO BEGIN
1132        TP:=TP+1;IF TP>U THEN TP:=0;
1142        IF TRACE[TP]>=0 THEN CODE(TRACE[TP]) END
1151      END;
1157 'K':BEGIN READ(I#);
1163       FOR J:=I TO I+6 DO
1172        WRITE('  ',S[J]#);CRLF
1185      END;
1186 'B':IF BP<BPLIM THEN BEGIN
1194       BP:=BP+1;WRITE(BP#,':');
1202       READ(BREAK[BP]#);CRLF END;
1207 'C':BEGIN (CLEAR BP)
1211       BP:=0;CRLF END;
1215 'Y':BEGIN FOR I:=1 TO BP DO
1226        WRITE('  ',BREAK[I]#);CRLF END;
1240 'E':BEGIN READ(P0#);CODE(P0) END;
1250 'U':IF P0>0 THEN BEGIN
1258       P0:=P0-1;CODE(P0) END;
1266 'N':BEGIN P0:=P0+1;CODE(P0) END;
1278 'Q':P:=-1
1283      ELSE BEGIN WRITE('??');CRLF END
1291   END (CASE OF CMD)
1292   UNTIL P<0;
1296   CRLF; WRITE(K#,' INSTR. EXECUTED.');CRLF
1319 END (MAIN).
INTERPRET(I), OR TRANSLATE(T)?
```

structure of the Pascal version is neat and highly readable, the debugging time is minimal. The Pascal source program is shown in listing 1. The program design is rather straightforward. Readers with some programming experience in any high level language should be able to read and understand it without the help of a flowchart or further explanation on program logic. Note that in the main program and procedure *exec*, the **case...of** statement is put to good use. In the BASIC version the interpreter commands have to be tested within a FOR loop by comparing the input character with a string array, and then an ON...GOTO statement is used to branch to various parts of the program.

It must be emphasized again that the interpreter executes p-codes and not Pascal statements. Therefore the user is required to have some knowledge of the p-machine and p-codes. In addition to this, the p-compiler should be instructed to list p-codes together with Pascal program statements during compilation. They will be cross-referenced when running the interpreter. Obviously this procedure is not as convenient and easy to use as an ordinary BASIC interpreter, but still it provides the only way for debugging Pascal programs in our present version. A new debugging scheme is being planned for the future which will enable the user to debug programs at the Pascal statement level. This means the user may refer to variables and arrays and statements rather than stack contents and p-code addresses. Part 2 will go into details of the design and implementation of the p-compiler.■

A "Tiny" Pascal Compiler

Part 2: The P-Compiler

Kin-Man Chung
124 Scottswood Dr
Urbana IL 61801

Herbert Yuen
POB 2591 Station A
Champaign IL 61820

When Niklaus Wirth introduced Pascal in 1971, one of the design objectives was to allow efficient program compilation. As far as we know, all existing Pascal compilers use the one pass compilation technique.

Newcomers to Pascal sometimes criticize features of the language such as declaring variables before use, and having constant and type declarations precede variable declarations. But such features are necessary to make a one pass compiler work (aside from the fact that it is also good programming practice to declare identifiers before use). Compared with multipass compilers, the job of writing a one pass compiler is relatively simple, since there is no need to store the program in its intermediate form.

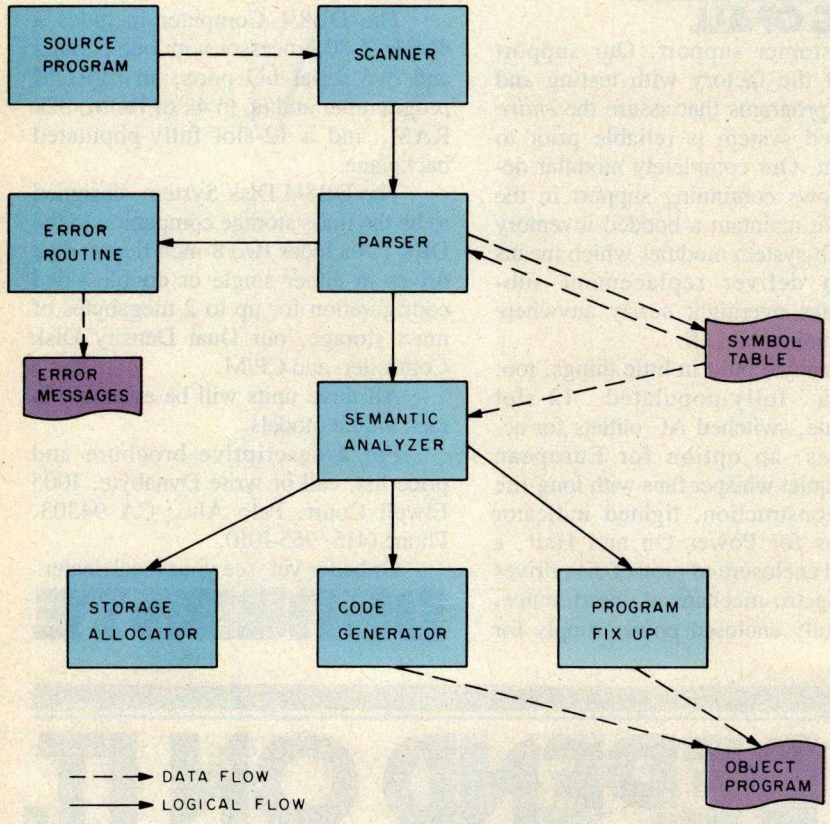


Figure 1: Logical arrangement and interconnections of the p-compiler modules.

Figure 1 shows the structure of our one pass Pascal compiler. The main portion is made up of the scanner, syntax analyzer, semantic analyzer and code generator. A brief overview of these functional portions of the compiler follows. Detailed descriptions will be given later.

The syntax analyzer is commonly called the *parser*. Its main function is to detect syntactical errors in the source program. The smallest unit of the source program that the parser looks at is called a *token*. For instance, the reserved word **while**, the symbol *:=*, or the identifier *idname* would be tokens. The main job of the scanner is to read the source program and output a token when needed by the parser. Irrelevant information such as blanks, comments and line boundaries are ignored.

To further simplify the work of the parser, the values of numeric constants are also evaluated by the scanner. The parser then parses the program according to the rules laid down by the syntax diagrams which were described in part 1 ("A Tiny Pascal Compiler," September 1978 BYTE, page 58) and generates error messages if illegal constructs are found. Identifier names are entered into a symbol table as they are declared. The symbol table is consulted by the parser as well as the semantic analyzer. After a Pascal construct is recognized, its meaning is analyzed by the semantic analyzer and appropriate p-codes are generated. Occasionally, there are forward references whose addresses cannot be determined at the time the codes are generated, but have to be resolved at a later time. Thus updates to the object program have to be done at the appropriate time.

This may sound complicated, but in fact a one pass compiler is actually the simplest compiler imaginable. The technique used by our parser is usually referred to as *top-down* parsing or goal oriented parsing. The top-down parsing algorithm assumes a general goal at the beginning. This goal is then broken down into one or more subgoals, depending on input strings and the rules in the syntax diagrams. The subgoals are realized by breaking them down into finer subgoals.

This is usually not a very efficient algorithm if backups are needed. The need for backups occurs if at some point we choose one subgoal from several others and find

Listing 1: BASIC version of the p-compiler. This program takes the Pascal program and compiles it into p-code. The term p-code stands for pseudocode, an assembler language code for a hypothetical computer which can be converted into an existing assembler language. Listing continues thru page 48.

```
10REM PASCAL SUBSET COMPILER FOR P-MACHINE
20REM BY KIN-MAN CHUNG
30REM 1/78. LAST VERSION 4/78.
40 N0=32\REM # OF RESERVED WORDS
50 T0=50\REM SYM TABLE SIZE
60 N1=32767\REM LARGEST INT
70 N2=8\REM IDENT LEN
80 DIM W0$(5*N0)\REM RESERVED WORDS
90 DIM T$(T0*N2)\REM SYMBOL TABLE
100 DIM T0$(T0)\REM KIND OF IDENT IN SYM TAB\C,V,P
110 DIM L$(64)\REM LINE BUFFER
120 DIM A$(N2),B$(5)
130 DIM S(100),S$(100)\REM STACKS
140 DIM T1(T0)\REM LEVEL OF ID IN SYM TBL
150 DIM T2(T0)\REM VAL(FOR CONST) OR ADR(FOR INT)OF ID IN S.T
160 DIM T3(T0)\REM ARRAY DIM OR# OF PROC PARAMETERS
170 W0$(1,40)="AND  ARRAYBEGINCALL CASE CONSTDIV  DO   "
180 W0$(41,80)="DOWNTELSE END  FOR  FUNC IF   INTEGMEM  "
190 W0$(81,120)="MOD  NOT  OF   OR   PROC READ REPEASHL  "
200 W0$(121,160)="SHR  THEN TO   TYPE UNTILVAR  WHILEWRITE"
210 DIM M$(27),C$(80)
220 M$="LITOPRLODSTOCALINTJMPJPCCSP"\REM P-CODE MNEMONICS
230 P8=1
240 P7=0\P9=P7\REM START CODE=0000
250 !"P-CODES STARTS AT 0000"
260 Q9=4096*2\REM LAST USABLE MEM
270 F5=-1
280 INPUT "WANT CODE PRINTED?",Y$
290 IF Y$="Y" THEN Y9=0 ELSE Y9=1
300 X$=" "\GOSUB 1240\REM GET A TOKEN
310 GOSUB 5340\REM BLOCK
320 Z=FNE1(".",9)
330 FILL P9,255\FILL P9+1,255\REM FILL IN EOF MARK
340 INPUT"INTERPRET(I), OR TRANSLATE(T)?",Y$
350 IF Y$="" THEN END
360 IF Y$="I" THEN CHAIN "INTERP"
370 IF Y$="T" THEN CHAIN "TRANS"
380 END
390REM **********
400REM ERROR ROUTINES
410REM ***
420REM FNE1..IF CURRENT TOKEN<>K$ THEN ERROR #E
430 DEF FNE1(K$,E)
440 IF S0$<>K$ THEN Z=FNE(E)
450 RETURN 0
460 FNEND
470REM ***
480REM FNE2..IF NEXT TOKEN<>K$ THEN ERROR #E
490 DEF FNE2(K$,E)
500 GOSUB 1240
510 IF S0$<>K$ THEN Z=FNE(E)
520 RETURN 0
530 FNEND
540REM ***
550REM PRINT ERROR MSG
560 DEF FNE(E9)
570 !TAB(C0+4),"↑",E9
580 GOSUB 610
590 STOP
600 RETURN 0\FNEND
610REM ERROR MSGS
620 ON INT((E9-1)/5)+1 GOTO 630,640,650,660,670,680,690,700
630 ON E9 GOTO 710,720,730,740,750
640 ON E9-5 GOTO 990,990,990,760,770
650 ON E9-10 GOTO 780,790,800,990,990
660 ON E9-15 GOTO 810,820,830,840,850
670 ON E9-20 GOTO 860,870,880,990,890
680 ON E9-25 GOTO 900,910,920,990,930
690 ON E9-30 GOTO 940,990,950,960,970
700 ON E9-35 GOTO 980
710 !"MEM FULL"\RETURN
720 !"CONST EXPECTED"\RETURN
730 !"'=' EXPECTED"\RETURN
740 !"IDENTIFIER EXPECTED"\RETURN
750 !"';' OR ':' MISSING"\RETURN
760 !"'.' EXPECTED"\RETURN
770 !"';' MISSING"\RETURN
780 !"UNDECLARED IDENT"\RETURN
790 !"ILLEGAL IDENT"\RETURN
800 !"':=' EXPECTED"\RETURN
810 !"'THEN' EXPECTED"\RETURN
820 !"';' OR 'END' EXPECTED"\RETURN
830 !"'DO' EXPEXTED"\RETURN
840 !"INCORRECT SYMBOL"\RETURN
850 !"RELATIONAL OPERATOR EXPECTED"\RETURN
860 !"USE OF PROC IDENT IN EXPR"\RETURN
870 !"')' EXPECTED"\RETURN
880 !"ILLEGAL FACTOR"\RETURN
```

after some processing that we have made the wrong choice. We would then have to undo what had been done by the wrong choice and back up to the point where we could try other alternatives. This is usually a messy business and involves a lot of bookkeeping. Fortunately, in the parsing of Pascal, no backup is necessary. A keyword is present at each decision point, and it determines what subgoal we should choose. An example will make this clear.

Suppose our goal is to recognize a *statement*. A statement can be a number of basic constructs: it can be an assignment statement, an **if** statement, a **case** statement or any other construct defined by the syntax diagram. The Pascal grammar is so designed that we know which type of statement we should choose by just looking at the next token. If the token is **if**, then we know it is going to be an if statement; if the token is **case**, it is going to be a case statement, etc. There would seem to be a problem if the token is an identifier, since the statement can be the beginning of an assignment statement or a procedure call. But this can be easily resolved by consulting the symbol table, where we also keep the attributes (data types, addresses, etc) of the identifiers. This is one of the reasons why identifiers and procedures must be declared before use: it makes compiler writing easier.

A top-down parser without backup can be implemented by using a technique called *recursive descent*. Such a parser uses a recursive procedure for each nonterminal in the syntax diagrams. A call is made to this procedure whenever a parse for such

Line Number	Remark
400	Error routines — FNE, FNE1, FNE2
1030	Get a character
1090	Input a line
1240	Get a token
1950	Enter entry into symbol table
2060	Search symbol table
2170	Constant declaration
2240	Get constant
2340	Variable declaration
2380	Simple expression
2610	Term
2850	Factor
3290	Expression
3490	Statement
5340	Block
6120	Push numeric
6150	Pop numeric
6180	Push string
6240	Pop string
6310	Code Generation — FNG
6520	Fixup forward references

Table 1: For easy reference the main subroutines of the p-compiler are listed here along with remarks regarding their uses.

```
890 !"'BEGIN' EXPECTED"\RETURN
900 !"'OF' EXPECTED"\RETURN
910 !"ILLEGAL HEX CONST"\RETURN
920 !"'TO' OR 'DOWNTO' EXPECTED"\RETURN
930 !"NUMBER OUT OF RANGE"\RETURN
940 !"'(' EXPECTED"\RETURN
950 !"'[' EXPECTED"\RETURN
960 !"']' EXPECTED"\RETURN
970 !"PARAMETERS MISMATCHED"\RETURN
980 !"DATA TYPE NOT RECOGNIZED"\RETURN
990 !"BUG"\RETURN
1000REM ********************
1010REM SCANNER
1020REM ********************
1030REM GETCHAR
1040 IF C0<L0 THEN 1060
1050 GOSUB 1090\GOTO 1040
1060 C0=C0+1\X$=L$(C0,C0)
1070 RETURN
1080REM **********
1090REM INPUT A LINE
1100 !%4I,C1," ",
1110 IF F5<0 THEN INPUT L$ ELSE 1160
1120 IF L$="" THEN 1100
1130 IF L$(1,1)="$" THEN 1210\REM MACRO FILE?
1140 L$=L$+" "\C0=0
1150 L0=LEN(L$)\RETURN
1160 IF TYP(F5)<>0 THEN 1190\REM EOF IF TYP=0
1170 CLOSE #F5\F5=F5-1\REM RETURN TO LAST ACTIVE FILE
1180 GOTO 1110
1190 READ #F5,L$\!L$
1200 GOTO 1130
1210 F5=F5+1\OPEN #F5,L$(2,LEN(L$))
1220 GOTO 1090
1230REM **********
1240REM GET A TOKEN
1250REM RETURN S0$=TOKEN, A$=STRING, N3=NUMERIC
1260 IF X$<>" " THEN 1280
1270 GOSUB 1030\GOTO 1260\REM FLUSH BLANKS
1280 IF X$<"A" THEN1460\REM INDENTIFIER?
1290 IF X$>"Z" THEN1460
1300 K=0\A$="                "
1310 IF K>=N2 THEN 1330\REM ONLY 1ST N2 LETTERS ARE USED
1320 K=K+1\A$(K,K)=X$
1330 GOSUB 1030
1340 T=ASC(X$)
1350 IF T>47 AND T<58 OR T>64 AND T<91 THEN 1310\REM DGT OR LTTR
1360REM BIN SERACH FOR RES WORDS
1370 I=1\J=N0%5-4
1380 B$=A$
1390 K=INT((I+J)/10)%5+1
1400 Z$=W0$(K,K+4)
1410 IF B$<=Z$ THEN J=K-5
1420 IF B$>=Z$ THEN I=K+5
1430 IF I<=J THEN 1390
1440 IF I-5>J THEN S0$=B$ ELSE S0$="IDENT"
1450 RETURN
1460 Z$=""
1470 IF X$<"0" THEN 1580\REM AN INTEGER?
1480 IF X$>"9" THEN 1580
1490 S0$="NUM"
1500 Z$=Z$+X$
1510 GOSUB 1030
1520 IF ASC(X$)>=48 AND ASC(X$)<=57 THEN 1500
1530 N3=VAL(Z$)
1540 IF N3<=N1 THEN RETURN
1550 E9=30\GOSUB 550
1560 N3=N1\RETURN
1570REM CHECK FOR SPECIAL SYMBOL
1580 IF X$<>":" THEN 1640
1590 GOSUB 1030
1600 IF X$="=" THEN 1620
1610 S0$=":"\RETURN
1620 S0$=":="
1630 GOSUB 1030\RETURN
1640 IF X$<>"<" THEN 1710
1650 GOSUB 1030
1660 IF X$=">" THEN 1690
1670 IF X$="=" THEN 1700
1680 S0$="<"\RETURN
1690 S0$="<>"\GOSUB 1030\RETURN
1700 S0$="<="\GOSUB 1030\RETURN
1710 IF X$<>">" THEN 1750
1720 GOSUB 1030\S0$=">"
1730 IF X$<>"=" THEN RETURN
1740 S0$=">="\GOSUB 1030\RETURN
1750 IF X$<>"'" THEN 1790
1760 S0$="STR"\C$=""
1770 GOSUB 1030\IF X$="'" THEN 1030
1780 C$=C$+X$\GOTO 1770
1790 IF X$<>"(" THEN 1820\REM IGNORE COMMENTS
1800 GOSUB 1030\IF X$<>")" THEN 1800
1810 GOSUB 1030\GOTO 1240
1820 IF X$<>"%" THEN 1930\REM HEX CONSTANT
1830 GOSUB 1030\S0$="NUM"\N3=0
1840 FOR I=1 TO 4
```

a nonterminal is required. It is easy to see why such a scheme would work. The stacking mechanism of the run time procedures ensures that we get back to the correct position in the syntax diagram after completing the parse of the nonterminal.

If you look at the syntax diagrams carefully, you will see that diagrams for certain nonterminals actually contain the nonterminal itself, either immediately or after several expansions. In terms of compiler writing this means that the procedures corresponding to these nonterminals would call themselves recursively.

BASIC Recursive Subroutines

Most versions of BASIC do not adequately support recursive subroutine calls. In North Star BASIC, the multiline function call can be invoked recursively, in a limited fashion. This is because the function parameters are local within the function definition and are pushed onto a stack when making a call.

The surprising fact is that most BASICs do not forbid a recursive call if one is made. For instance, the following BASIC subroutine, which is an inefficient way of printing the first N integers in descending order, is probably permitted in most BASICs:

```
100 PRINT N
200 IF N=0 THEN RETURN
300 N=N-1
400 GOSUB 100
500 RETURN
```

The problem of doing recursive calls in BASIC is that of preserving the values of the identifiers in the subroutines. This can be done by using a stack. The values of the identifiers are pushed onto the stack before a recursive call, and popped out of the stack in the reverse order when returning from the call. In BASIC, the stack can be simulated by an array:

```
10 DIM S(100)
11 P=0
12 REM INITIALIZE STACK POINTER
.
.
.
1000REM PUSH X INTO STACK
1010 S(P)=X
1020 P=P+1
1030 RETURN
2000REM POP X FROM STACK
2010 P=P-1
2020 X=S(P)
2030 RETURN
```

```
1850 T=ASC(X$)
1860 IF T>=48 AND T<=57 THEN 1880
1870 IF T>=65 AND T<=70 THEN T=T-7 ELSE 1910
1880 I=T-48
1890 N3=N3*16+T\GOSUB 1030\NEXT
1900 RETURN
1910 IF I>1 THEN Z=FNE(27)
1920 S0$="%"\RETURN
1930 S0$=X$\GOTO 1030
1940REM **********
1950REM ENTER SYMBOL INTO TABLE
1960 T1=T1+1
1970 T$((T1-1)*N2+1,T1*N2)=A$
1980 T0$(T1,T1)=K$\REM STORE TYPE
1990 IF K$<>"C" THEN 2010
2000 T2(T1)=N3\RETURN\REM STORE VALUE
2010 T1(T1)=L1\REM STORE LEVEL OF IDENT
2020 IF K$<>"V" THEN RETURN
2030 IF NOT P9 THEN RETURN\REM SP WAS ALLOCATED FOR PROC PARS
2040 T2(T1)=D0\D0=D0+1\RETURN\REM STORE OFFSET
2050REM **********
2060REM FIND IDENT A$ IN T$,STARTING FROM T1 AND UP
2070REM RETURN POINTER TO TABLE IF FOUND, ELSE RETURN 0
2080 J=(T1-1)*N2+1
2090 FOR I=T1 TO 1 STEP -1
2100 IF A$=T$(J,J+N2-1) THEN EXIT 2130
2110 J=J-N2\NEXT
2120 I=0
2130 RETURN
2140REM ********************
2150REM PARSER AND CODER
2160REM ********************
2170REM CONSTANT DECLARATION
2180 Z=FNE1("IDENT",4)
2190 Z=FNE2("=",3)
2200 GOSUB 1240\GOSUB 2240
2210 K$="C"\GOSUB 1950
2220 GOTO 1240
2230REM **********
2240REM CONSTANT
2250 IF S0$="NUM" THEN RETURN
2260 IF S0$="IDENT" THEN 2290\REM CONST?
2270 Z=FNE1("STR",2)
2280 N3=ASC(C$)\RETURN\REM TAKE 1ST CHAR
2290 GOSUB 2060\IF I=0 THEN FNE(2)
2300 IF T0$(I,I)<>"C" THEN FNE(2)
2310 N3=T2(I)\RETURN
2320 GOTO 1240
2330REM **********
2340REM VARIABLE DECLARATION
2350 Z=FNE1("IDENT",4)
2360 K$="V"\GOSUB 1950\GOTO 1240
2370REM **********
2380REM SIMPLE EXPRESSION
2390 IF S0$="+" THEN 2420
2400 IF S0$<>"-" THEN 2590
2410 Y$=S0$\GOSUB 6180
2420 GOSUB 1240
2430 GOSUB 2610
2440 GOSUB 6240
2450 IF Y$="-" THEN Z=FNG(1,0,1)
2460 IF S0$="+" THEN 2500
2470 IF S0$="-" THEN 2500
2480 IF S0$="OR   " THEN 2500
2490 RETURN
2500 Y$=S0$\GOSUB 6180
2510 GOSUB 1240
2520 GOSUB 2610
2530 GOSUB 6240
2540 IF Y$="-" THEN 2570
2550 IF Y$="+" THEN 2580
2560 Z=FNG(1,0,14)\GOTO 2460
2570 Z=FNG(1,0,3)\GOTO 2460
2580 Z=FNG(1,0,2)\GOTO 2460
2590 GOSUB 2610\GOTO 2460
2600REM **********
2610REM TERM
2620 GOSUB 2850
2630 IF S0$="*" THEN 2700
2640 IF S0$="DIV  " THEN 2700
2650 IF S0$="AND  " THEN 2700
2660 IF S0$="MOD  " THEN 2700
2670 IF S0$="SHL  " THEN 2700
2680 IF S0$="SHR  " THEN 2700
2690 RETURN
2700 Y$=S0$\GOSUB 6180\REM PUSH
2710 GOSUB 1240\GOSUB 2850
2720 GOSUB 6240
2730 IF Y$="DIV  " THEN 2790
2740 IF Y$="MOD  " THEN 2800
2750 IF Y$="*" THEN 2810
2760 IF Y$="SHL  " THEN 2820
2770 IF Y$="SHR  " THEN 2830
2780 Z=FNG(1,0,15)\GOTO 2630\REM "AND"
2790 Z=FNG(1,0,5)\GOTO 2630
2800 Z=FNG(1,0,7)\GOTO 2630
```

One important part missing from our compiler is the ability to recover from errors. Of course all syntactical errors are caught by our compiler and somewhat meaningful messages are printed to indicate errors. However, if an error is found, the compiler is aborted prematurely and will not resume compiling. Such a compiler is, of course, not acceptable in practice. But with the understanding that this compiler will be used as a bootstrap compiler, as discussed in part 1, it is tolerable. A compiler with simple error recoveries would not be particularly difficult to implement but would involve a lot of programming codes and processing time. We hesitate to add things to an already big and slow program.

It is generally difficult to implement a compiler with sophisticated error recovery features. Such a compiler would not only detect errors, but would also try to repair the damages caused by such errors. The compiler has to make some assumptions about the nature of the errors and the intention of the author. This is usually difficult.

If our concern is solely that of locating all errors in a single parse of the source program, there are simple ways of doing it. Upon detecting an error, the compiler simply skips the input text until it can safely resume the compilation process. To do this the compiler looks for certain keywords or *stopping* symbols for hints to resume the parsing process. For instance, if we find an error while parsing a conditional expression, we skip the input tokens and search for symbols, such as =, > =, etc, and keywords such as **then** and **do** or perhaps **begin**. If we do this for all the parts of the language constructs, we will at least have a compiler that would resume compilation after an error is encountered in the hope of finding all syntactic errors in one pass, and which would give meaningful diagnostics for most errors.

To reduce the size of the program shown in listing 1, comments are kept to a minimum. Each module or subroutine is clearly identified. To facilitate easy reference, the important subroutines and variables are shown in table 1 and table 2, respectively.

Scanner and Symbol Table Management

Each time the p-compiler calls the scanner (line 1260, listing 1), the input text is scanned and a new token is produced. This is done by calling a subroutine (line 1040) that returns a character from the input string. Since the input/output (IO) routines are line oriented instead of character oriented, a line buffer (L$) is used to

```
2810 Z=FNG(1,0,4)\GOTO 2630
2820 Z=FNG(1,0,17)\GOTO 2630
2830 Z=FNG(1,0,18)\GOTO 2630
2840REM **********
2850REM FACTOR
2860 IF S0$="IDENT" THEN 2940
2870 IF S0$="NUM" THEN 3060
2880 IF S0$="STR" THEN 3080
2890 IF S0$="(" THEN 3100
2900 IF S0$="MEM   " THEN 3140
2910 IF S0$="NOT   " THEN 3260
2920 Z=FNE(23)
2930REM *** IDENTIFIER
2940 GOSUB 2060
2950 IF I=0 THEN Z=FNE(11)
2960 IF T0$(I,I)="P" THEN Z=FNE(21)\REM PROC NAME
2970 IF T0$(I,I)<>"Y" THEN 3000
2980 Z=FNG(5,0,1)\REM FUNC
2990 I=I-1\GOTO 4290\REM T2(I)=ADD OF FUNC
3000 IF T0$(I,I)="A" THEN 3190\REM ARRAY
3010 IF T0$(I,I)<>"C" THEN 3030
3020 Z=FNG(0,0,T2(I))\GOTO 1240\REM CONST
3030 Z=FNG(2,L1-T1(I),T2(I))\REM ID
3040 GOTO 1240
3050REM *** NUMERIC CONST
3060 Z=FNG(0,0,N3)\GOTO 1240
3070REM *** STRNG CONST
3080 Z=FNG(0,0,ASC(C$))\GOTO 1240
3090REM *** PAREN EXPR
3100 GOSUB 1240\GOSUB 3290
3110 IF S0$=")" THEN 1240
3120 Z=FNE(22)\RETURN
3130REM *** READ MEMORY
3140 Z=FNE2("[",33)
3150 GOSUB 1240\GOSUB 3290
3160 Z=FNE1("]",34)
3170 GOSUB 1240
3180 Z=FNG(2,255,0)\RETURN
3190 X=I\GOSUB 6120
3200 Z=FNE2("[",33)
3210 GOSUB 1240\GOSUB 3290
3220 Z=FNE1("]",34)
3230 GOSUB 6150\Z=FNG(18,L1-T1(X),T2(X))
3240 GOTO 1240
3250REM *** NEGATE
3260 GOSUB 1240\GOSUB 2850
3270 Z=FNG(1,0,16)\RETURN
3280REM **********
3290REM EXPRESSION
3300 GOSUB 2390\REM SIMPLE EXP
3310 IF S0$="=" THEN 3380
3320 IF S0$="<>" THEN 3380
3330 IF S0$="<" THEN 3380
3340 IF S0$="<=" THEN 3380
3350 IF S0$=">" THEN 3380
3360 IF S0$=">=" THEN 3380
3370 RETURN
3380 Y$=S0$\GOSUB 6180\REM PUSH
3390 GOSUB 1240\GOSUB 2390
3400 GOSUB 6240\REM POP
3410 IF Y$="=" THEN Z=FNG(1,0,8)
3420 IF Y$="<>" THEN Z=FNG(1,0,9)
3430 IF Y$="<" THEN Z=FNG(1,0,10)
3440 IF Y$=">=" THEN Z=FNG(1,0,11)
3450 IF Y$=">" THEN Z=FNG(1,0,12)
3460 IF Y$="<=" THEN Z=FNG(1,0,13)
3470 RETURN
3480REM **********
3490REM STATEMENT
3500 IF S0$="IDENT" THEN 3630
3510 IF S0$="IF    " THEN 4440
3520 IF S0$="FOR   " THEN 5170
3530 IF S0$="WHILE" THEN 4800
3540 IF S0$="CASE " THEN 4890
3550 IF S0$="REPEA" THEN 4730
3560 IF S0$="BEGIN" THEN 4590
3570 IF S0$="READ " THEN 4040
3580 IF S0$="WRITE" THEN 3870
3590 IF S0$="MEM   " THEN 4650
3600 IF S0$="CALL " THEN 4240
3610 RETURN
3620REM *** ASSIGNMNT
3630 GOSUB 2060
3640 IF I=0 THEN Z=FNE(11)
3650 IF T0$(I,I)="A" THEN 3700\REM ARRAY
3660 IF T0$(I,I)="V" THEN 3760\REM INT VAR
3670 IF T0$(I,I)="Y" THEN 3760\REM FUNC RETURN VALUE
3680 IF T0$(I,I)="P" THEN 4290\REM PROC CALL
3690 Z=FNE(12)
3700 X=I\GOSUB 6120\REM PUSH TBL ADD
3710 X=16\GOSUB 6120\REM INDEX ADD MODE
3720 Z=FNE2("[",33)
3730 GOSUB 1240\GOSUB 3290
3740 Z=FNE1("]",34)
3750 GOTO 3780
3760 X=I\GOSUB 6120
```

hold a line, and a counter (C0) is used to indicate the character just read. When the end of a line is reached, the line input routine (line 1100) is called to read in a new line.

In our compiler we also provide the capability of invoking or recalling a file of Pascal text from disk. This is initiated by a command that starts with a dollar sign ($) in the first column followed immediately by the name of the disk file to be inserted and compiled. Since North Star BASIC allows four disk files to be open at the same time, there can be four levels of file nesting. The variable F5 is used to indicate this level. If it is equal to −1, then input is taken from the keyboard. The initial input is from the keyboard. This feature is quite useful, since we can store procedures that are commonly used in a disk library, and have them recalled when needed.

Usually, the token that the scanner returns is a number that represents the token class the symbol is in. To make the program more readable, we use string variable S0$. Possible values returned by the scanner are: ; , :=, BEGIN, IDENT, and NUM. The last two tokens, which are tokens for identifiers and numbers, require some further information. A$ and N3 are also used to store the textual representation of the identifier and the value of the number, respectively.

The recognition of a valid token is a straightforward process and will not be detailed here. Since : and := are both valid tokens, the scanner, after seeing the : , must also look at the next character to determine the correct token. This can be done by using a one character look ahead. When the scanner is entered, a character is assumed to have been read, and upon exit from the scanner, a character beyond the current token is read.

Another problem that the scanner may have is that of recognizing reserved words. The reserved words are stored in a table in sorted order. When an identifier is found, it is compared with the entries in the table, by performing a binary search. If it is not in the table, it is assumed to be a user defined identifier.

In Pascal programs, identifiers are declared at the beginning of each procedure block. The scope of an identifier covers the entire block containing it (and any of the blocks inside that block). A simple symbol management scheme that reflects such scope rules makes use of a stack. When the compiler enters a procedure block, a segment of the stack is used to store identifiers for the block. If the procedure block contains another procedure block, then another

```
3770 X=0\GOSUB 6120
3780 GOSUB 1240
3790 IF S0$=":=" THEN 3810
3800 Z=FNE(13)\GOTO 3820
3810 GOSUB 1240
3820 GOSUB 3290\GOSUB 6150
3830 K=X\GOSUB 6150
3840 Z=FNG(3+K,L1-T1(X),T2(X))
3850 RETURN
3860REM *** WRITE
3870 Z=FNE2("(",31)
3880 GOSUB 1240\IF S0$<>"STR" THEN 3950
3890 L=LEN(C$)\IF L>1 THEN 3910
3900 Z=FNG(0,0,ASC(C$))\Z=FNG(8,0,1)\GOTO 3940
3910 FOR I=1 TO L
3920 Z=FNG(0,0,ASC(C$(I,I)))\NEXT
3930 Z=FNG(0,0,L)\Z=FNG(8,0,8)
3940 GOSUB 1240\GOTO 4000
3950 GOSUB 3290\K=1
3960 IF S0$="#" THEN K=3\REM DEC
3970 IF S0$="%" THEN K=5\REM HEX
3980 IF K>1 THEN GOSUB 1240
3990 Z=FNG(8,0,K)
4000 IF S0$="," THEN 3880
4010 Z=FNE1(")",22)
4020 GOTO 1240
4030REM *** READ
4040 Z=FNE2("(",31)
4050 Z=FNE2("IDENT",4)
4060 GOSUB 2060\IF I=0 THEN Z=FNE(11)
4070 X=I\GOSUB 6120
4080 IF T0$(I,I)="A" THEN 4190
4090 IF T0$(I,I)="V" THEN L=0 ELSE Z=FNE(4)
4100 GOSUB 1240\K=0
4110 IF S0$="#" THEN K=2\REM DEC
4120 IF S0$="%" THEN K=4\REM HEX
4130 Z=FNG(8,0,K)
4140 IF K>0 THEN GOSUB 1240
4150 GOSUB 6150\Z=FNG(L+3,L1-T1(X),T2(X))
4160 IF S0$="," THEN 4050
4170 Z=FNE1(")",31)
4180 GOTO 1240
4190 Z=FNE2("[",33)
4200 GOSUB 1240\GOSUB 3290
4210 Z=FNE1("]",34)
4220 L=16\GOTO 4100
4230REM *** ABSOLUTE MEM CALL
4240 Z=FNE2("(",31)
4250 GOSUB 1240\GOSUB 3290
4260 Z=FNE1(")",22)
4270 Z=FNG(4,255,0)\GOTO 1240
4280REM *** PROC OR FUNC CALL
4290 K2=0\K3=I
4300 IF T3(I)=0 THEN 4400\REM NO PARAMETER
4310 Z=FNE2("(",31)
4320 X=K2\GOSUB 6120
4330 X=K3\GOSUB 6120
4340 GOSUB 1240\GOSUB 3290
4350 GOSUB 6150\K3=X
4360 GOSUB 6150\K2=X\K2=K2+1
4370 IF S0$="," THEN 4320
4380 IF K2<>T3(K3) THEN Z=FNE(35)
4390 Z=FNE1(")",22)
4400 Z=FNG(4,L1-T1(K3),T2(K3))
4410 IF K2 <>0 THEN Z=FNG(5,0,-K2)
4420 GOTO 1240
4430REM *** IF
4440 GOSUB 1240
4450 GOSUB 3290
4460 Z=FNE1("THEN ",16)
4470 GOSUB 1240
4480 X=C1\GOSUB 6120\REM FORWARD REF POINT
4490 Z=FNG(7,0,0)\REM JPC
4500 GOSUB 3490
4510 IF S0$<>"ELSE " THEN 6520
4520 GOSUB 6150\K=X
4530 X=C1\GOSUB 6120
4540 Z=FNG(6,0,0)\REM JMP
4550 X=K\GOSUB 6540\REM FIXUP FORWD REF
4560 GOSUB 1240\GOSUB 3490
4570 GOTO 6520
4580REM *** COMPOUND STTMNT
4590 GOSUB 1240
4600 GOSUB 3490
4610 IF S0$=";" THEN 4590
4620 IF S0$="END  " THEN 1240
4630 Z=FNE(17)\RETURN
4640REM *** WRITE MEM
4650 Z=FNE2("[",33)
4660 GOSUB 1240\GOSUB 3290
4670 IF S0$<>"]" THEN Z=FNE(34)
4680 Z=FNE2(":=",13)
4690 GOSUB 1240\GOSUB 3290
4700 Z=FNG(3,255,0)
4710 RETURN
4720REM *** REPEAT .. UNTIL
```

segment of the stack on top of the existing segments is used for identifiers of this block. After successful compilation of a procedure, its segment of the stack can be discarded, since there is no further use for this part of the symbol table. In this way, we can also eliminate possible interference with identifiers in some other blocks. We also see that since the block delimiting mechanism is hierarchical, use of stack is also appropriate. Figure 2 illustrates two-level block nesting.

Readers may have noticed the similarities between this symbol table stacking scheme and the run time storage allocation scheme discussed in part 1. Since the symbol table deals with a static structure, it is much simpler.

Within the segment of the symbol table for a procedure block, further data structures can be set up for storing the identifiers. We chose to use what we feel is the simplest method: store the identifiers sequentially, in their order of appearance. This means that search also has to be done sequentially. Since most procedures have only a small number of identifiers, this should work well in most cases. Other more sophisticated structures such as a balanced binary tree or hashed table are commonly used in larger compilers.

The symbol table also contains some information about the identifiers. The identifier type has to be kept with the symbol table. Specific information is needed

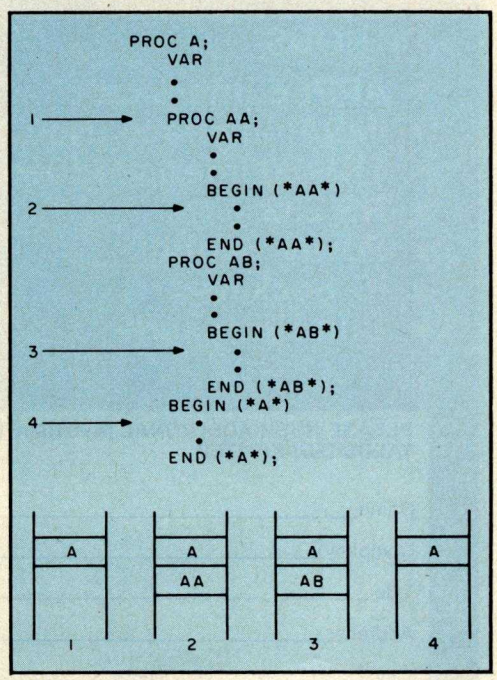


Figure 2: Example symbol table at various points of compilation.

```
4730 X=C1\GOSUB 6120
4740 GOSUB 1240\GOSUB 3490
4750 IF S0$=";" THEN 4740
4760 Z=FNE1("UNTIL",10)
4770 GOSUB 1240\GOSUB 3290
4780 GOSUB 6150\Z=FNG(7,0,X)\RETURN
4790REM *** WHILE .. DO
4800 GOSUB 1240\X=C1\GOSUB 6120
4810 GOSUB 3290\X=C1\GOSUB 6120
4820 Z=FNG(7,0,0)
4830 Z=FNE1("DO   ",18)
4840 GOSUB 1240\GOSUB 3490
4850 GOSUB 6150\K=X\GOSUB 6150
4860 Z=FNG(6,0,X)
4870 X=K\GOTO 6540
4880REM *** CASE .. OF
4890 GOSUB 1240\GOSUB 3290
4900 Z=FNE1("OF   ",25)
4910 I2=1\REM # OF CASE STATMNTS
4920 I1=0\REM # OF CASE LABELS
4930 GOSUB 1240\GOSUB 2240
4940 Z=FNG(1,0,21)\Z=FNG(0,0,N3)\Z=FNG(1,0,8)
4950 GOSUB 1240\IF S0$=":" THEN 4990
4960 Z=FNE1(",",5)
4970 X=C1\GOSUB 6120\Z=FNG(7,1,0)\REM A MATCH FOUND?
4980 I1=I1+1\GOTO 4930
4990 K=C1\Z=FNG(7,0,0)\REM GOTO NEXT CASE STMNT IF NO MATCH
5000 FOR I=1 TO I1\GOSUB 6520\NEXT\REM FIXUP FORWD REFS
5010 X=K\GOSUB 6120
5020 GOSUB 1240\X=I2\GOSUB 6120
5030 GOSUB 3490\GOSUB 6150\I2=X
5040 IF S0$="ELSE " THEN 5090
5050 IF S0$<>";" THEN 5130
5060 K=C1\Z=FNG(6,0,0)\REM EXIT AFTER A CASE STMNT
5070 GOSUB 6520
5080 X=K\GOSUB 6120\I2=I2+1\GOTO 4920
5090 K=C1\Z=FNG(6,0,0)\GOSUB 6520
5100 X=K\GOSUB 6120
5110 GOSUB 1240\X=I2\GOSUB 6120
5120 GOSUB 3490\GOSUB 6150\I2=X
5130 Z=FNE1("END  ",17)
5140 FOR I=1 TO I2\GOSUB 6520\NEXT\REM FIXUP FORWD REFS
5150 Z=FNG(5,0,-1)\GOTO 1240\REM POP VAL OF CASE EXP
5160REM *** FOR
5170 Z=FNE2("IDENT",4)
5180 GOSUB 3630\GOSUB 6120
5190 F9=1\IF S0$="TO   " THEN 5210\REM REMEMBER UP OR DOWN
5200 Z=FNE1("DOWNT",28)\F9=0
5210 GOSUB 1240\GOSUB 3290
5220 GOSUB 6150\K=X\X=C1\GOSUB 6120
5230 Z=FNG(1,0,21)\Z=FNG(2,L1-T1(K),T2(K))
5240 Z=FNG(1,0,13-F9-F9)\X=C1\GOSUB 6120\Z=FNG(7,0,0)
5250 X=F9\GOSUB 6120\X=K\GOSUB 6120
5260 Z=FNE1("DO   ",18)\GOSUB 1240
5270 GOSUB 3490\GOSUB 6150\Z=FNG(2,L1-T1(X),T2(X))
5280 K=X\GOSUB 6150\Z=FNG(1,0,20-X)
5290 Z=FNG(3,L1-T1(K),T2(K))
5300 GOSUB 6150\K=X\GOSUB 6150\Z=FNG(6,0,X)
5310 X=K\GOSUB 6540
5320 Z=FNG(5,0,-1)\RETURN\REM POP OFF VAL OF LOOP CNTRL VAR
5330REM **********
5340REM BLOCK
5350 D0=3\REM RESERVED FOR STATIC LINK,DYNAMIC LINK & RETN ADD
5360 T2(T1-K1)=C1\REM INIT ADD OF THE PROC BLOCK
5370 Z=FNG(6,0,0)\REM JMP TO STARTING BLK ADD
5380 X=T1-K1\GOSUB 6120
5390 IF S0$="CONST" THEN 5460
5400 IF S0$="VAR  " THEN 5550
5410 IF S0$="PROC " THEN 5730
5420 IF S0$="FUNC " THEN 5770
5430 IF S0$="BEGIN" THEN 5980
5440 Z=FNE(25)
5450REM *** CONST DCL
5460 GOSUB 1240
5470 GOSUB 2170
5480 Z=FNE1(";",5)\GOSUB 1240
5490 IF S0$="VAR  " THEN 5550
5500 IF S0$="PROC " THEN 5730
5510 IF S0$="FUNC " THEN 5770
5520 IF S0$="BEGIN" THEN 5980
5530 GOTO 5470
5540REM *** VARIABLE DCL
5550 L=0\F9=1
5560 GOSUB 1240\GOSUB 2340
5570 L=L+1\IF S0$="," THEN 5560
5580 Z=FNE1(":",5)
5590 GOSUB 1240\IF S0$="ARRAY" THEN 5610
5600 Z=FNE1("INTEG",36)\GOTO 5670
5610 Z=FNE2("[",33)\GOSUB 1240\GOSUB 2240
5620 Z=FNE2("]",34)\Z=FNE2("OF   ",26)\Z=FNE2("INTEG",36)
5630 D0=D0-L
5640 FOR I=T1-L+1 TO T1
5650 T0$(I,I)="A"\T3(I)=N3+1
5660 T2(I)=D0\D0=D0+N3+1\NEXT
```

for each type of identifier. For constants, the information is the values of the constants; for program variables, the information is the address pair (level, offset from base address); for procedures and functions, it is the address pairs and the number of parameters; and, lastly, for array variables, the information is the address pair as well as array sizes. See table 2 for actual variables that are used to store these quantities.

The symbol table is used by both the parser and the semantic analyzer. The information in the symbol table is used in a number of ways. The type of identifier is used, for instance, to check the type consistency in an expression. When a variable is referenced or a procedure or function called, the symbol table is searched to obtain the level and relative address from the base address. The number of parameters in a procedure or function is used to check the correct matching of parameters in actual procedure or function calls.

An identifier is searched for by starting from the end of the symbol table and working towards the beginning. (Viewing the table as a stack, we say that we search from the *top* of the stack down to the *bottom*.) There are two reasons for this searching direction. First, identifiers in the current block are more likely to be referenced and should be searched first. Secondly, suppose that a variable X is declared in both an outer and an inner block: by searching for X from top to bottom of the stack, we can be sure that we will find X of the inner block first, in accordance with the scope rule.

Parser, Semantic Analyzer, and Coder

The parser, the semantic analyzer and the coder are not separate routines, but are intermixed in a large routine. In most cases, after the successful parsing of a statement, its meaning is also understood by the compiler. Thus the semantic analyzer either requires minimal extra processing or is implicit in the parser and disappears altogether.

The parser, as we have mentioned before, uses a top-down technique called recursive descent. Since there is a close correspondence between the parser and the syntax diagrams of the Pascal grammar, there should be no difficulties in understanding the parsing process. The parser adopts the convention of one token look ahead which is similar to the one character look ahead convention used by the scanner. The variable S0$ is used to hold the next token to be read by the parser.

There is a part of the Pascal grammar, commonly referred to as the dangling

```
5670 Z=FNE2(";",5)
5680 GOSUB 1240\IF S0$="PROC " THEN 5730
5690 IF S0$="FUNC " THEN 5770
5700 IF S0$="BEGIN" THEN 5980
5710 L=0\F9=1\GOSUB 2340\GOTO 5570
5720REM *** PROC DCL
5730 Z=FNE2("IDENT",4)
5740 K1=0\K$="P"\GOSUB 1950
5750 L1=L1+1\GOTO 5810
5760REM *** FUNC DCL
5770 Z=FNE2("IDENT",4)
5780 K$="F"\GOSUB 1950\REM FUNC ADDRSS
5790 L1=L1+1\K1=1
5800 K$="Y"\GOSUB 1950\REM FUNC VALUE
5810 K2=K1\GOSUB 1240
5820 X=T1\GOSUB 6120
5830 X=D0\GOSUB 6120
5840 IF S0$<>"(" THEN 5890
5850 GOSUB 1240\F9=0\GOSUB 2340\K1=K1+1
5860 IF S0$="," THEN 5850
5870 Z=FNE1(")",22)
5880 GOSUB 1240\T3(T1-K1)=K1-K2
5890 Z=FNE1(";",5)
5900 FOR I=1 TO K1\REM FUNC VALUE & PARS HAVE - OFFSET
5910 T2(T1-I+1)=-I\NEXT
5920 GOSUB 1240\GOSUB 5340\L1=L1-1
5930 GOSUB 6150\D0=X
5940 GOSUB 6150\T1=X
5950 Z=FNE1(";",5)
5960 GOSUB 1240\GOTO 5410
5970REM *** START OF EXECUTIBLE STTMNTS
5980 GOSUB 1240\GOSUB 6150\K=X
5990 X=T2(K)\GOSUB 6540
6000 T2(K)=C1\REM START BLOCK ADDR
6010 Z=FNG(5,0,D0)
6020 GOSUB 3490
6030 IF S0$<>";" THEN 6050
6040 GOSUB 1240\GOTO 6020
6050 IF S0$<>"END. " THEN Z=FNE(17)
6060 GOSUB 1240
6070 Z=FNG(1,0,0)
6080 RETURN
6090REM ***********
6100REM END PARSER AND CODER
6110REM ***********
6120REM PUSH X INTO STACK
6130 S(S9)=X\S9=S9+1\RETURN
6140REM **********
6150REM POP X FROM STACK
6160 S9=S9-1\X=S(S9)\RETURN
6170REM **********
6180REM PUSH Y$ INTO STACK
6190 L=LEN(Y$)
6200 S$(P8,P8+L-1)=Y$
6210 X=P8\GOSUB 6120\REM PUSH START & END STRNG POS
6220 X=P8+L-1\GOSUB 6120
6230 P8=P8+L\RETURN
6240REM POP Y$ FROM STACK
6250 GOSUB 6150
6260 L=X\GOSUB 6150
6270 Y$=S$(X,L)
6280 P8=P8-L+X-1
6290 RETURN
6300REM **********
6310REM GENERATE CODES
6320 DEF FNG(X1,X2,X3)
6330 B$="    "
6340 FILL P9,X1\FILL P9+1,X2
6350 FILL P9+2,FNA(X3)\FILL P9+3,FNB(X3)
6360 IF Y9 THEN 6400\REM IF INPUT FROM KEYBOARD THEN DONT ECHO
6370 IF X1<16 THEN 6390
6380 B$(1,1)="X"\X1=X1-16\REM INDEX
6390 !%4I,C1," ",M$(X1*3+1,X1*3+3),B$,%3I,X2,%6I,X3
6400 C1=C1+1\P9=P9+4
6410 IF P9>=Q9 THEN Z=FNE(1)
6420 RETURN 0
6430 FNEND
6440REM **********
6450 DEF FNB(Z)
6460 N=INT(Z/256)
6470 IF N<0 THEN N=256+N
6480 RETURN N
6490 FNEND
6500 DEF FNA(Z)=Z-INT(Z/256)*256
6510REM **********
6520REM FIXUP FORWORD REF
6530 GOSUB 6150
6540 N=P7+X*4
6550 FILL N+2,FNA(C1)\FILL N+3,FNB(C1)
6560 IF Y9 THEN RETURN
6570 !"ADD AT",X," CHANGED TO",C1
6580 RETURN
READY
```

else, that is ambiguous. The statement:

```
if cond1 then if cond2 then stat1 else stat2;
```

can be parsed in two ways. The **else** statement can be associated with the first **if** or with the second **if**, producing entirely different results.

We resolve this difficulty by always associating the **else** statement with the most recent **if**. If an **else** statement with the first **if** is desired, one of these two methods should be used:

```
if cond1 then
   if cond2 then stat1 else
else stat2;
```

or:

```
if cond1 then begin
   if cond2 then stat1
   end
else stat2;
```

The situation is similar to the **case** statement with the added feature of an optional **else** statement. If the statement for the last **case** label is an **if** statement, we then have the dangling **else** problem. This is resolved in the same manner.

There are three functions used to print messages when errors are detected. The function FNE(X) prints the error message corresponding to error code X. FNE1(A$,X) checks to see if the current token is equal to A$, and prints the error message corresponding to error code X if not. FNE2 is similar to FNE1 except that the scanner is first called to get a new token. As we mentioned earlier, the compiler aborts as soon as an error is found. Therefore these error routines do not return to the calling procedure.

The code generator requires more work: care must be taken to store important values in stacks due to the inability of BASIC to fully support recursive subroutine calls. Otherwise the coder is more or less straightforward, since the p-codes are so designed (see part 1) that there is a direct correspondence between simple Pascal statements and p-codes. Table 3 shows the almost direct translation of Pascal statements into p-codes.

The declarative statements (**const**, **var**, **proc**, and **func**) do not produce any executable statements; they merely provide information about declared identifiers. The first executable code encountered when entering a procedure or function block is a forward jump instruction to the main body of the block. This jump is necessary since in general there may be procedures and func-

```
P-CODES START AT 0000
WANT CODE PRINTED?N
   0 ?$LST2.2
   0 CONST CR=13;LF=10;
   1 VAR A,B,C,D:INTEGER;
   1 FUNC MAX4(X1,X2,X3,X4); {LARGEST OF 4 NUMBERS}
   1    FUNC MAX2(X1,X2); {LARGEST OF 2 NUMBERS}
   2       BEGIN
   3          IF X1>X2 THEN MAX2:=X1
   9          ELSE MAX2:=X2
  12          END;
  14    BEGIN
  14       MAX4:=MAX2(MAX2(X1,X2),MAX2(X3,X4))
  28       END;
  30 BEGIN
  30    REPEAT
  31       READ (A#,B#,C#,D#);
  39       WRITE ('THE LARGEST IS',MAX4(A,B,C,D)#,CR,LF)
  67       UNTIL A<0
  69    END.
INTERPRET(I), OR TRANSLATE(T)?N
READY
LOAD DECODE
READY
RUN
   0  JMP   0    30  JMP   0    14  JMP   0     3  INT   0     3
   4  LOD   0    -2  LOD   0    -1  OPR   0     >  JPC   0    11
   8  LOD   0    -2  STO   0    -3  JMP   0    13  LOD   0    -1
  12  STO   0    -3  OPR   0   RET  INT   0     3  INT   0     1
  16  INT   0     1  LOD   0    -4  LOD   0    -3  CAL   0     3
  20  INT   0    -2  INT   0     1  LOD   0    -2  LOD   0    -1
  24  CAL   0     3  INT   0    -2  CAL   0     3  INT   0    -2
  28  STO   0    -5  OPR   0   RET  INT   0     7  CSP   0 INNUM
  32  STO   0     3  CSP   0 INNUM  STO   0     4  CSP   0 INNUM
  36  STO   0     5  CSP   0 INNUM  STO   0     6  LIT   0    84
  40  LIT   0    72  LIT   0    69  LIT   0    32  LIT   0    76
  44  LIT   0    65  LIT   0    82  LIT   0    71  LIT   0    69
  48  LIT   0    83  LIT   0    84  LIT   0    32  LIT   0    73
  52  LIT   0    83  LIT   0    14  CSP   0 OUTST  INT   0     1
  56  LOD   0     3  LOD   0     4  LOD   0     5  LOD   0     6
  60  CAL   0    14  INT   0    -4  CSP   0 OUTNM  LIT   0    13
  64  CSP   0 OUTCH  LIT   0    10  CSP   0 OUTCH  LOD   0     3
  68  LIT   0     0  OPR   0     <  JPC   0    31  OPR   0   RET
```

Listing 2: Sample Pascal program with compiled p-code. The number at the beginning of each source line is the offset of the corresponding p-code from the base address.

Variable Name	Remark
A$	String of the token returned by the scanner
C0	Input buffer pointer
C1	P-code address pointer
D0	Run time storage counter
E9	Error code
F5	Active input file unit number; keyboard=−1
K1	Number of parameter in the previous block
L0	Length of the input line
L1	Static level of procedure
L$	Input line buffer
M$	P-code mnemonics
N0	Reserved word table size
N1	Largest integer
N2	Length of identifier name
N3	Numeric value of token (token = "NUM") or ASCII value of string (token = "STR")
P8	Stack pointer for S$
P9	P-code absolute memory address counter
S	Stack for numeric values
S9	Stack pointer for S
S$	Stack for strings
S0$	Next token
T0	Symbol table size
T1	Symbol table pointer
T$	Symbol table: identifier
T0$	Symbol table: type of identifier V: variable A: array C: constant P: procedure F: function Y: parameter
T1( )	Symbol table: level
T2( )	Symbol table: value (constant) or displacement (variable) or address (proc or func)
T3( )	Symbol table: array size (array) or number of parameter (proc or func)
X	Value to be pushed or popped
X$	Next character to be read by the scanner
Y$	String to be pushed or popped
W0$	Table for reserved words

Table 2: Important variables used in the p-compiler.

tions whose codes take up space. The second executable code of the block increments the stack pointer (INT). This allocates space for the triplet (static link, dynamic link and return address) plus any variables declared. The number of spaces for the variables is already known from the declaration portion of the procedure block. The variable D0 is used to keep track of the space to be allocated at the activation of the block.

Note that no space is allocated for constants. If a constant is referenced, a load literal (LIT) instruction is generated instead of a load (LOD) instruction. Also note that the procedure or function parameters and the function return value do not reserve any space in the procedure or function block called. Space is reserved before the call is made. Therefore, these values have negative displacement from the base address of the called procedure or function.

When a call is made to a function, the space for function return value is allocated by incrementing the stack pointer (line 2980 in listing 1) (this step is skipped for a procedure call). The parameter expression is then evaluated (line 4250), putting the resultant value on the stack. Thus, space is allocated for each parameter and initialized with the value of the parameter expression. Upon return from a procedure, the stack pointer is decremented by an amount equal to the space allocated

Pascal source		p-codes	
*x+10*y[5]*		LOD	X
		LIT	10
		LIT	5
		LODX	Y
		OPR	*
		OPR	+
a:=exp;		(exp)	
		STO	A
if *exp* **then** *stm1* **else** *stm2;*		(exp)	
		JPC	0,1b1
		(stm1)	
		JMP	1b2
	1b1	(stm2)	
	1b2	. . .	
for *i:=exp1* **to** *exp2* **do** *stm;*		(exp1)	
		STO	I
		(exp2)	
	1b1	OPR	CPY
		LOD	I
		OPR	>=
		JPC	0,1b2
		(stm)	
		LOD	I
		OPR	INC
		STO	I
		JMP	1b1
	1b2	INT	−1
while *exp* **do** *stm;*	1b1	(exp)	
		JPC	0,1b2
		(stm)	
		JMP	1b1
	1b2	. . .	
case *exp* **of**		(exp)	
c1b1,c1b2:stm1;		OPR	CPY
c1b3 :stm2;		LIT	c1b1
else *stm3*		OPR	=
end;		JPC	1,1b1
		OPR	CPY
		LIT	c1b2
		OPR	=
		JPC	0,1b2
	1b1	(stm1)	
		JMP	1b4
	1b2	OPR	CPY
		LIT	c1b3
		OPR	=
		JPC	0,1b3
		(stm2)	
		JMP	1b4
	1b3	(stm3)	
	1b4	INT	−1
repeat *stm* **until** *exp;*	1b1	(stm)	
		(exp)	
		JPC	0,1b1
i:=funca(exp1,exp2);		INT	1
		(exp1)	
		(exp2)	
		CAL	funca
		INT	−2

Table 3: Code generation for various Pascal constructs. For readability, the p-codes are given in assembly form. The italic identifiers in the Pascal statements are nonterminals that can be substituted by any valid expansion. The codes for these quantities are represented by parenthesized identifiers.

for the parameters, getting back to the state before the procedure call. Upon returning from a function call, the stack pointer is also decremented by the same amount, but since a space has been allocated before the function call, the function return value is now on top of the stack, ready for further processing. This simple scheme works very efficiently and should lower the overhead usually associated with procedure or subroutine calls.

Listing 2 gives an output from the compiler for a Pascal program that prints out the maximum of four numbers. There are of course better ways of writing the program, but it does illustrate some ideas of the compiler discussed so far.

There is no optimization of the p-codes produced. Limited optimization can be done on the local level, and some optimization is actually done in the p-code to machine code translator. The problem of producing efficient codes is a difficult one, and is not addressed properly in our project. Given the simplicity of the p-machine and p-code, the p-compiler is efficient. But whether the combination of p-compiler and translator produces efficient 8080 code is uncertain.

This completes our discussion of the p-compiler. In part 3 (see November 1978 BYTE), we give a detailed discussion of a translator for converting the p-code into executable 8080 machine code.■

REFERENCES

1. Wirth, N, "The Programming Language Pascal," *Acta Informatica,* 1, pages 35 thru 63, 1971.

2. Jensen, K, and Wirth, N, *Pascal: User Manual and Report* (second edition) Springer Verlag, New York, 1974.

3. Wirth, N, *Algorithms + Data Structures = Programs,* Chapter 5, "Language Structures and Compilers," Prentice-Hall, Englewood Cliffs NJ, 1976.

A "Tiny" Pascal Compiler

Part 3: P-Code to 8080 Conversion

Kin-Man Chung
124 Scottswood Dr
Urbana IL 61801

Herbert Yuen
POB 2591 Station A
Champaign IL 61820

In part 1 of this series (September 1978 BYTE, page 58) we defined a Pascal subset language in terms of syntax diagrams. The p-machine and its instruction set and a p-code interpreter were also described. In part 2 (October 1978 BYTE, page 34) we presented the design and implementation of the p-compiler. The subject matter for this part is the translation of p-codes to executable 8080 machine codes. We will also discuss the implementation of run time support routines and code optimization.

Compiler-Interpreter Systems

To understand why we need a p-code to 8080 translator, we should first take a brief look at the different structures of compiler-interpreter systems. The most widely used structure for microcomputers is the *interpreter.* Since interpreters are written in the target computer's assembly language, their memory size is small. They are self-contained in the sense that they include an editor for creating source programs and run time routines to do all computations. Memory storage for source programs is also small. The only disadvantage is speed. Execution time for a typical BASIC program is estimated to be about 300 to 1000 times the execution time of the same program written in assembly language. Interpreters may spend more than 70 percent of their time scanning source symbols character by character, parsing the syntax and checking errors. No matter how many times a program statement is executed, the parsing procedure is repeated every time.

This problem can be readily solved by separating the parsing and execution steps. Before execution, the source program is compiled and intermediate code is generated. Thus scanning and parsing are done only once for each program statement. This is the so-called *compiler-interpreter* scheme used in some BASIC compilers. Execution of the intermediate codes is by interpretation. The gain in speed over a pure interpreter is a factor of approximately 2 to 10. However, the gain in speed is paid for by extra memory storage needed for intermediate codes.

The *compile-go* and *compile-link-go* approaches are commonly used for many high level language compilers in mainframe computer systems. These compilers generate relocatable binary codes. The compile-link-go approach has the advantage of linking together different modules of programs that are compiled separately, such as those in a subroutine library. This is done by a linking loader. However, due to limited system resources like memory and peripheral devices in microcomputers, these two structures are rarely used. Further, since Pascal is designed for fast compilation, linkage of program modules may be done at the source language level.

Among those four structures just mentioned, the compiler-interpreter seems to be most appropriate for implementation on microcomputers. However, execution speed is still slow because intermediate codes are interpreted rather than executed directly by the computer. An obvious solution to this problem is to translate the intermediate codes into executable machine codes. Thus, each intermediate code is decoded once by a program which we call a *translator.* The translated machine code can be expected to run about two to five times faster than interpreted intermediate codes. Therefore, the overall gain in speed, compared with a pure interpreter, is a factor of approximately 10 to 50. (Preliminary test runs in

our system show that Pascal programs run about 15 times faster than the same program written in BASIC.) We call this structure *compile-translate-go.*

The five compiler-interpreter structures we discussed above are summarized in table 1. The compile-go and compile-translate-go are rather similar in structure. Compile-go actually combines the process of compiling and generating executable codes into one step. The binary codes are generated by straightforward algorithms without optimization, because code optimization would require more complex program logic and make the compiler even larger. Separating compilation and translation into two steps significantly reduces the size of the compiler. Local optimization techniques can also be applied during translation. Code optimization will be discussed later. Since p-codes are designed to be machine independent to make the compiler portable, the translator is responsible for producing efficient codes for a target computer.

Designing the Run Time Routines

Run time routines form an essential part of all compiler-interpreter systems in microcomputers. Large computers can do fixed point, floating point and decimal arithmetic with 32 bit or larger word sizes in single instructions. Many microcomputers, on the other hand, can do only basic integer arithmetic with 8 bit words (bytes). Therefore, multiple instructions are needed to implement 16 bit operations like multiply, divide, subtract, logical operations and multibit shifts. The run time routines, sometimes referred to as *run time support package*, are a collection of subroutines written in assembly language that can be called by an interpreter or any program to perform various arithmetic and logical operations. Usually they include subroutines for IO conversion between ASCII and binary data.

The design of run time routines for our compiler system is based on three principles:

- **Fast implementation and clarity**: A straightforward approach is followed so that the overall package can be debugged and tested quickly and modified easily.
- **Speed**: The best known algorithms are used for computer arithmetic to achieve fastest execution speed possible. However, tricks such as self-modifying code are not used.
- **Memory storage**: The package is expected to be fairly compact. Since p-codes are translated mostly into subroutine calls, the number of instructions to set up arguments to be passed to the subroutine should also be minimal.

As described in part 1, the p-machine has a data stack and four registers: stack pointer T, base register B, program counter P, instruction register I. Since the translator takes care of the program counter and p-code instructions are not needed after translation, all we need are the stack pointer and base register. In the current version of our run time routines, contiguous memory storage is used to represent the data stack. For the sake of program clarity and easy debugging, the 8080 machine stack is not used, although using it for dual purposes as a data stack and temporary storage for normal program logic is possible and probably more efficient.

Figure 1 shows the structural differences between the p-machine stack which we implement and the 8080 machine stack. Since

Table 1: Summary of different structures of compiler-interpreter systems.

Structure	Example	Step	Input	System software	Output	Remarks
interpreter	BASIC, APL interpreter	1	source program	interpreter (execution)		Most popular for microcomputers. Advantage: conserves memory space. Disadvantage: very slow execution speed.
compiler-interpreter	BASIC-E, Pascal compiler	1 2	source program intermediate code	compiler interpreter (execution)	intermediate code	The interpreter may overlay the compiler to save memory space. Advantage: faster execution speed.
compile-go	WATFIV,PL/C compiler	1	source program	compiler	executable code	Only used in large computers. Disadvantage: size is too big for microcomputers.
compile-link-go	FORTRAN IV, PL/I, COBOL compiler	1 2	source program binary code	compiler linking loader	binary code executable code	Widely used in large computers. Advantage: fast execution speed. Disadvantage: requires more system resources.
compile-translate-go	Pascal compiler (by authors)	1 2	source program p-code	compiler translator	p-code executable 8080 code	Advantage: size of compiler is reduced, fast execution speed, increased portability, easy implementation.

integer data is stored as pairs of 8 bit bytes (character strings are stored as single dimensional arrays, two bytes to each element and only the low order byte is used; see descriptions in part 1), each load instruction increments the stack pointer by 2. The order of the byte pair is arranged as high-low because it is more convenient to use than low-high. The stack pointer always points to the low order byte of the 16 bit integer, which is on top of the stack.

Register pair D,E is dedicated for use as the stack pointer, while registers H and L are mainly used for 16 bit operations such as DAD, LHLD, SHLD and PCHL. When needed, register pairs D,E and H,L can be easily exchanged using the XCHG instruction. Since the base address remains unchanged within a procedure block, a 2 byte fixed memory location (with symbolic name BB) is used to represent the base register. The LHLD and SHLD instructions are used to retrieve and update the base address value. A summary of register assignments for implementation of the p-machine is shown in table 2.

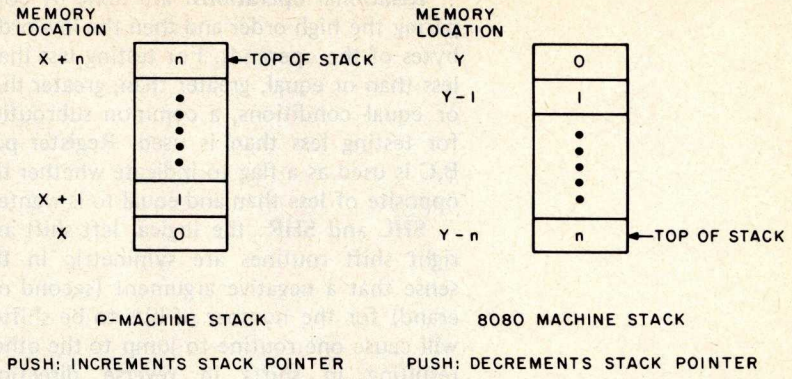


Figure 1. Differences between p-machine and 8080 stacks. This figure shows n+1 entries on each of the stacks.

P-machine	8080 run time routines
P: program counter	PC
T: stack pointer	D,E register pair
B: base register	memory location BB (16 bits)
I: instruction register	—
data stack	memory storage

Table 2: Register and storage assignment for runtime routines.

Coding the Run Time Routines

Most of the subroutines are easily understandable. The routines for load, store, call and load constant are coded by direct translation from the interpreter program to 8080 assembly language, keeping in mind that each stack element (one data item) occupies two bytes. The routines for arithmetic and logical operations and IO conversions require more programming effort. In general, single operand functions such as negate, logical not and increment are performed one byte at a time in register A. Double operand operations such as add, divide and logical or are performed with register pairs H,L and B,C. The entire runtime package occupies about 1 K bytes of memory. The following are remarks on coding some of the not-so-trivial subroutines.

PUSH and **POP**: for most double operand functions, subroutine POP is called first to get the two operands from the stack (memory) and put in register pairs H,L (first operand) and B,C (second operand). After the operations, subroutine PUSH is called to put the result from H,L back onto the stack.

Add and **subtract**: since DAD (double precision add) is the only 8080 instruction for double operand 16 bit operation, subtraction is done by adding the 2's complement of the second operand to the first. A message will be issued if overflow occurs and execution continues without any corrective action. The condition for overflow is detected by the rule:

if [sign(arg.1) ⊕ sign(arg.2) ⊕ carry ⊕ sign (result)] = 1, then overflow; otherwise nothing.

MULT16: 16 bit signed multiplication is done in two stages using an 8 bit multiplication routine. First, multiply the second operand by the high order byte of the first operand; the result is in register pair H,L. Second, continue the multiplication (left shift and double add) with the low order byte of the first operand; the result is in register pair H,L. This method is very efficient. In comparison, conventional 16 bit multiplication routines require more PUSH, POP and XCHG instructions because there are not enough registers to shift two 16 bit words and also update a loop counter. Overflows are ignored, as this is the usual practice for integer multiplication.

DIV16: 16 bit signed division is one of the most difficult routines to implement. First the signs of both operands are saved on a stack and are then converted to positive integers (actually the divisor is made negative in 2's complement because subtraction is done with a double add instruction). The divisor is also checked for zero value, and if so, a DIVIDE CHECK message is issued and the routine returns. Division is carried out as a sequence of subtraction and shifts. At the end, the signs of the quotient and remainder are corrected according to the original signs of the operands. The same routine is also used for calculation of the MOD function.

Relational operations: are done by comparing the high order and then the low order bytes of the operands. For testing less than, less than or equal, greater than, greater than or equal conditions, a common subroutine for testing less than is used. Register pair B,C is used as a flag to indicate whether the opposite of less than and equal to is wanted.

SHL and SHR: the logical left shift and right shift routines are symmetric in the sense that a negative argument (second operand) for the number of bits to be shifted will cause one routine to jump to the other, resulting in shifts in reverse direction.

INNUM: the conversion subroutine for input integers allows leading zeros and blanks and may optionally be preceded by a plus or minus sign (+ or −). It also checks for the absolute magnitude of the integer, which must be less than 32,768.

OUTNUM: conversion of binary integers to ASCII is done by repeated division by 10. The 16 bit divide routine is utilized.

P-code Translation

In general, p-codes are translated to subroutine call instructions which jump to the appropriate entry points in the run time routines. Output from the translator is an 8080 machine language program containing mostly subroutine call instructions. Some p-codes, such as load and store, require additional instructions to set up the arguments to be passed. Address offsets are always placed in register pair B,C and the static level difference is placed in register A. The jump instruction in p-code simply becomes a JMP instruction in 8080 with the correct address determined by the translator. The p-code addresses in CAL and JPC instructions are similarly taken care of by the translator. The complete list of 8080 code corresponding to each p-code is shown in table 3.

Hexadecimal Op code	P-code	8080 Mnemonic	Commentary
00	LIT 0,n	LXI B,n CALL LIT	
01	OPR 0,0	JMP P00;	procedure return routine
	OPR 0,n	CALL Pn ;	one of the 21 arithmetic/logical routines
02	LOD v,d		
	a) v=0	LXI B,2d CALL LOD	
	b) v>0	LXI B,2d MVI A,v CALL LOD1	
	c) v=255	CALL LODA;	load absolute address
12	LODX v,d		
	a) v=0	LXI B,2d CALL LODX	
	b) v>0	LXI B,2d MVI A,v CALL LODX1	
03	STO v,d		
	a) v=0	LXI B,2d CALL STO	
	b) v>0	LXI B,2d MVI A,v CALL STO1	
	c) v=255	CALL STOA;	store absolute address
13	STOX v,d		
	a) v=0	LXI B,2d CALL STOX	
	b) v>0	LXI B,2d MVI A,v CALL STOX1	
04	CAL v,a		
	a) v=0	CALL CAL JMP x	
	b) v>0	MVI A,v CALL CAL1 JMP x	
	c) v=255	CALL CALA;	machine language subroutine interface
05	INT 0,n	LXI H,2n CALL INT	
06	JMP 0,a	JMP x	
07	JPC 0,a	LDAX D; DCX D DCX D; RAR ; JNC x	get conditional code decrement stack pointer test conditional code
	JPC 1,a		(same as JPC 0,a except JC x)
08	CSP 0,n (n=0...5)	CALL SYSn;	one of the 6 conversion routines
	for n=8:		(output a string)
	LIT 0,c_1 LIT 0,c_2 . . . LIT 0,c_n LIT 0,n CSP 0,8	MVI C,n; CALL SYS8 DB c_1 DB c_2 . . . DB c_n	# of char.

Table 3: P-code to 8080 translation. LIT, LOS, STOX1, INT, LODA, etc, are used as symbolic entry points in the runtime routines. There are 22 routines for the OPR instructions: P00, P01, . . ., P21. There are seven standard routines for IO conversion: SYS0, SYS1, . . ., SYS5 and SYS8. The variable x is used as the memory address in the translated 8080 code corresponding to p-code address a in a call and jump instruction.

The 2 Pass Translator

The structure of the translator is similar to that of the interpreter. Both programs read p-codes from memory and decode them. The interpreter calls a simulator to execute the p-codes. The translator writes translated 8080 code in memory. The major difference between them is that the translator needs three additional tables to keep track of p-code and 8080 addresses. Since all p-code addresses are relative to the starting p-code of the program, the program is relocatable. The memory address corresponding to p-code address for any backward and forward referenced jumps can be calculated easily because all p-codes are four bytes long. The number of 8080 instructions generated per p-code is also not constant as shown in table 3. Therefore, it is necessary to build a table of 8080 addresses corresponding to p-code addresses to be used in jump and call instructions. However, it is not practical to build a table of 8080 addresses for every p-code because it will take too much memory storage for large programs. Only the addresses of those p-codes that are being referenced need be entered into the table.

P-code to 8080 machine code translation is done in two passes. During the first pass, p-code addresses in CAL, JMP and JPC instructions are entered into a table. The table is sorted after the completion of the first pass. Actual translation is carried out in the second pass. P-codes are fetched one by one from memory and decoded. The address of each p-code is checked with those in the address table. If it indicates that the current p-code is being referenced, the current 8080 address is entered to the corresponding 8080 address table. Then 8080 machine codes are produced according to the translation rules shown in table 3.

For CAL, JMP and JPC instructions, the p-code address in the instruction is looked up in the address table using a binary search. If the corresponding 8080 address has already been entered, it is output in the translated code; otherwise it is a forward referenced address. When the latter case occurs, it is necessary to record the current 8080 address in a *forward reference* table. Then, instead of the 8080 address (which is not yet known), its position in the table is output in the translated code. At the end of the second pass the forward referenced addresses are fixed up by the following procedure:

a) Get the 8080 address from the forward reference table (call it P).
b) Get the table entry (call it J) at address P in the translated program.
c) Get the updated 8080 address (call it A) at table entry J.

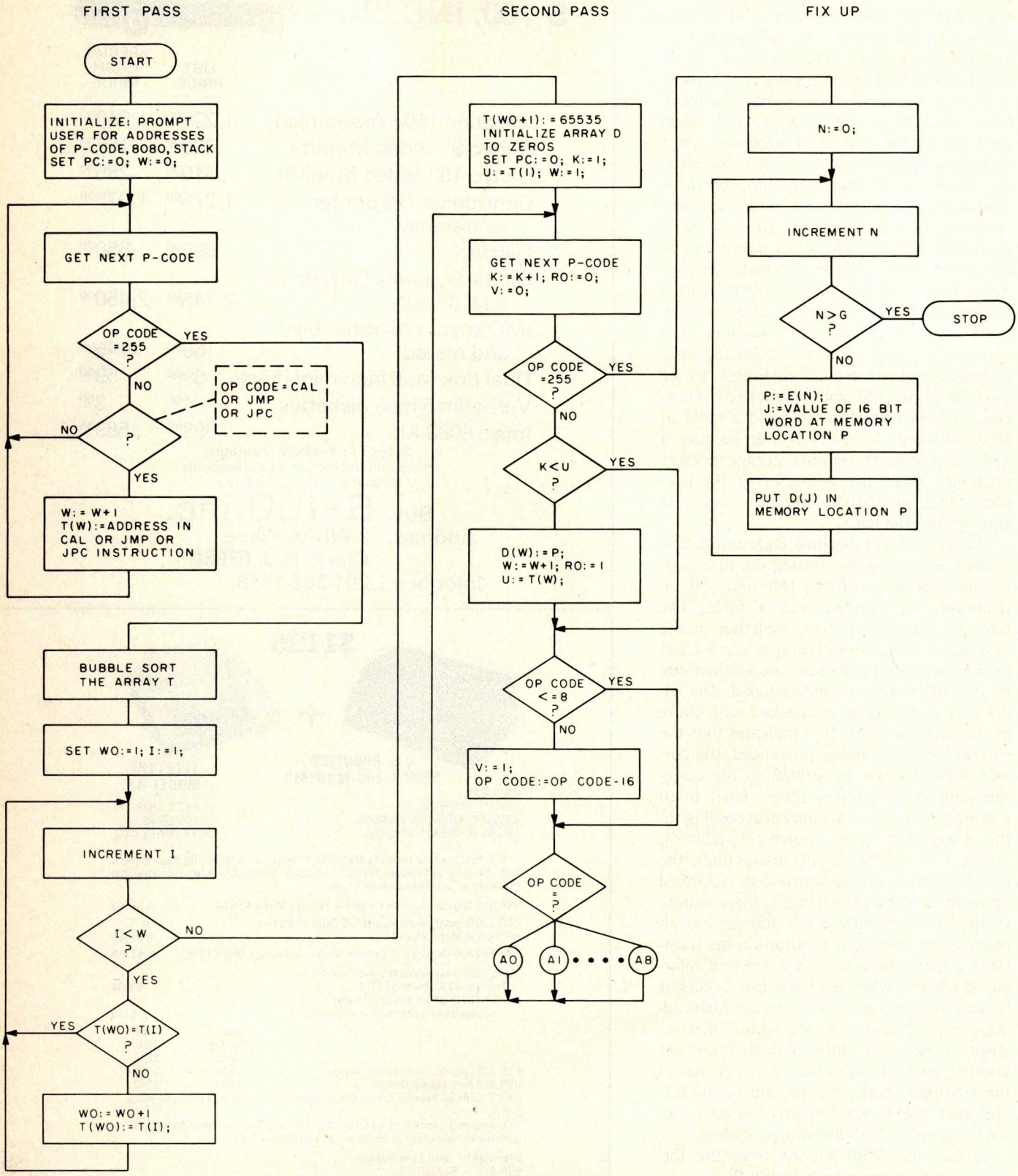


Figure 2: A simplified flowchart of the translator. A0, A1, . . .A8 are program segments for generating 8080 code for the p-code with peephole optimization as illustrated by the rules in table 4. Refer to table 5 for a description of the variables.

d) Write the correct address A back to memory location P.

Figure 2 is a simplified flowchart of the translator. The part for code generation is not shown, but it can be easily understood by referring to tables 3 and 4. Table lookup is done by binary search through the sorted table. The table elements are entered sequentially during the first pass. A simple bubble sort algorithm is used to sort the table. This method works fine for small Pascal programs. For larger programs, and thus more referenced addresses, the bubble sort algorithm is too slow because the number of comparisons is of order n^2 for n elements. A binary tree sorting algorithm with order $n \log n$ will be used for our next version of the translator.

The various entry points in the runtime routines are initialized in the translator as a series of string constants. These hexadecimal addresses are converted to integers and placed in arrays so they can be accessed very easily later on.

When execution begins, the program prompts the user for starting addresses of the p-code program, the output 8080 code, and starting and ending addresses of the data stack. The following three instructions are generated to initialize the data stack and pointer:

LXI H,STK1	starting address of data stack.
LXI D,STK2	2's complement of stack ending address.
CALL #1A00	runtime routine (initialization)

The program then begins its first pass. The number of address references and actual number of referenced addresses are displayed at the end of the first pass. During the second pass, cross references of p-code and 8080 addresses, which may be useful for future references, are listed in hexadecimal form. At the end of the translation, sizes of the p-code program and 8080 code are displayed.

Code Optimization

Code optimization is a technique employed by most compilers to improve the object code produced. Many sophisticated code optimization techniques are known today but are outside the scope of this article. We shall describe only one form of local optimization technique which is being used in our project. Local optimization

Table 4: Summary of peephole optimization. The goal is to reduce the size of the object program. The optimized code is more efficient than the unoptimized 8080 code. For the redundant store fix, the load instruction cannot be referenced elsewhere in the program.

Source of optimization	Example	P-code	8080 code	Optimized 8080 code
Redundant jump instructions	beginning of a procedure without inner procedure	n: JMP 0,n+1	JMP x	no code generated
Redundant loads and stores	J:=J+5; A[J]:=X;	STO v,d * LOD v,d	(as usual) (as usual)	(as usual) INX D; increment stack INX D; pointer
Repeated load of the same variable	A[J]:=A[J]+Y;	LOD v,d LOD v,d	(as usual) (as usual)	(as usual) CALL P21; copy
INT instruction with small constant	procedure call without parameter	INT 0,0	LXI H,#0000 CALL INT	no code generated
	procedure call	INT 0,n (−3 ⩽ n ⩽ 2)	LXI H,2n CALL INT	INX D } (repeat n times, n > 0) INX D } DCX D } (repeat n times, n < 0) DCX D }
Load negative constants	B:= −20;	LIT 0,n OPR 0,1	LXI B,n CALL LIT CALL P01	LXI B,−n CALL LIT
Add and subtract small constants (n ⩽ 3)	array subscripts A[J+2]:= B[K−1]:= L:=L+1;	LIT 0,n OPR 0,2 LIT 0,n OPR 0,3	LXI B,n CALL LIT CALL P02 LXI B,n CALL LIT CALL P03	CALL P19; increment (repeat n times) CALL P20; decrement (repeat n times)
Load zeros	P:=0;	LIT 0,0	LXI B,#0000 CALL LIT	XRA A INX D STAX D INX D STAX D

*Must be an unreferenced p-code

is done within a straight line block of code with no jumps into or out of the middle of the block. *Peephole* optimization is one form of local optimization which examines only small pieces of object code.

Since most code optimization techniques are difficult to build in a syntax directed code generation algorithm, peephole optimization is particularly useful in improving the intermediate code. Each improvement may lead to opportunities for further improvements. The technique can be applied repeatedly to get maximum optimization. In our translator, peephole optimization is applied only once during the second pass.

The goal of optimization is to minimize the size of the translated 8080 code and to increase execution speed without sacrificing a lot of time during translation. The peephole technique is quite simple. It examines only a single code or two consecutive codes. Some redundant p-codes are obvious and can be easily recognized. For example, the JMP instruction generated at the beginning of a procedure block which does not contain inner blocks is redundant. Similarly, the p-code INT 0,0 (increment stack pointer) generated after a procedure call with no arguments can be eliminated. The biggest benefit comes from optimizing redundant load and store instructions, because they are relatively slow in the current implementation. For example, a LOD instruction immediately following a STO instruction of the same variable can be replaced by an increment stack pointer instruction, because the variable is still on the stack. However, if the LOD instruction has a label, ie: is being referenced somewhere in the program, we cannot be sure that the STO instruction is always executed immediately before the LOD instruction.

Other sources of peephole optimization are the replacement of specific operations by more efficient instructions. Addition and subtraction of small constants (less than 4) occur frequently in array subscripts and loop counters. They can be replaced by repeated increment or decrement instructions. Some p-codes are translated into in line 8080 code instead of a call to runtime routines. Table 4 is a summary of peephole optimization used in the translator. Note that the optimized code always takes less memory space than the unoptimized code.

An Example

The various modules of the compiler system have been described. Now let us look at a complete program example. Listing 1 shows the compilation, translation and exe-

cution of a sample Pascal program. The program is stored in a disk file with file name T4. It is a sorting program that uses a binary tree algorithm. As mentioned before, it is more efficient than a bubble sort algorithm. The two subroutines in this program will be used in our next version of the translator (written in Pascal). The main program begins by asking the user to input an integer K (K must be less than 110) for the number of items to be sorted. It then reads the K+1 bytes of data starting from hexadecimal memory location 1A00 (the location where runtime routines are stored). The data items are read one at a time and procedure ENTER is called to build a binary tree with these items. Procedure TRAV is then called to traverse the tree recursively in the "left subtree..root..right subtree" fashion and the data with sorted order is placed in array S. Finally, array S is printed.

The p-compiler generates 145 p-codes (0 to 144) for this program. Afterwards, it uses a CHAIN statement (North Star BASIC) to load the translator program from disk, and overlays the compiler. The translator begins by asking the user to input memory addresses of runtime routines, p-code program, output 8080 code and data stack. At the end of the first pass, 20 address references are recorded. After sorting, it is found that there are only 15 actual labels. Output from the second pass of the translator is a cross-reference of p-code program counter and memory addresses of the corresponding translated 8080 code. The leftmost column is the p-code program counter. Hexadecimal memory addresses are printed in groups of 15 per line. With the exception of the first one, only the two low order hexadecimal digits are printed. At the end of the second pass, 11 forward references are recorded. A total of 766 bytes of 8080 code are generated. Compared to the size of the p-code program, the translated code is 1.32 times larger. This ratio usually ranges between 1.05 and 1.35, depending on program structure and the types of statements used.

After translation is completed, control is transferred to the disk operation system (DOS). The runtime routines are loaded from the disk file, PAS.LIB, to hexadecimal memory location 1A00. Then execution may begin by typing a JPxxxx command (jump to xxxx), where xxxx is the starting hexadecimal memory address of the translated code. In listing 1, two separate runs are shown: the first one sorts eight numbers (K+1 with K = 7) and the second sorts 21 numbers. The user may get back to BASIC by typing JP2A04, where 2A04 is the entry point of BASIC. (The command !CHR$(129)

T$	–	table of p-code address labels
D$	–	table of 8080 address corresponding to address labels in array T$
E	–	table of forward references
W	–	count of address references
WO	–	count of actual labels
G	–	count of forward references
K	–	p-code instruction counter
X	–	memory location of current p-code
P	–	8080 program counter of the translated code
F	–	current op code
V	–	=1 means indexed load or store
RO	–	=1 means current p-code is being referenced
U	–	program counter of the next referenced p-code

Table 5: Table of important variables and arrays in the translator program shown in flowchart form in figure 2.

```
P-CODES STARTS AT 0000
WANT CODE PRINTED?N
   0 ?$T4
   0  ( PGM -- SORTING BY BINARY TREE )
   0 VAR I,J,K,N,NEW:INTEGER;
   1 T,L,R,S:ARRAY[110] OF INTEGER;
   1  *
   1 PROC ENTER(N);
   1  VAR J:INTEGER;
   2  BEGIN J:=0;
   5   REPEAT
   5    IF N<=T[J] THEN
   9      IF L[J]<>0 THEN J:=L[J]
  17      ELSE BEGIN L[J]:=NEW;J:=0 END
  24    ELSE IF R[J]<>0 THEN J:=R[J]
  32         ELSE BEGIN R[J]:=NEW;J:=0 END
  39   UNTIL J=0;
  43   T[NEW]:=N;NEW:=NEW+1
  48  END;
  51
  51 PROC TRAV(J);  ( TRAVERSE THE TREE )
  51  BEGIN IF L[J]<>0 THEN TRAV(L[J]);
  62   S[K]:=T[J];K:=K+1;
  70   IF R[J]<>0 THEN TRAV(R[J])
  79  END;
  80
  80 BEGIN (MAIN)
  80  T[0]:=255;NEW:=0;
  86  READ(K#);WRITE(13,10);
  92  FOR I:=0 TO K DO BEGIN
  99   L[I]:=0;R[I]:=0; ENTER(MEM[I+%1A00]) END;
 116  K:=0; TRAV(0);
 121  FOR I:=0 TO K-1 DO WRITE(' ',S[I]#);
 140  WRITE(13,10)
 144 END.
INTERPRET(I), OR TRANSLATE(T)?T

*** P-CODE TO 8080 TRANSLATION ***
ADDR (HEX) OF PAS.LIB:1A00
ADDR (HEX) OF P-CODE:0000
ADDR (HEX) OF OUTPUT 8080 PGM:0800
STACK START ADDR (HEX):5000
STACK END ADDR (HEX):7FFF
 20 REFERENCES
 15 ACTUAL LABELS

   0  0809 0C 0C 12 17 1D 23 29 31 34 3B 41 49 4E 51
  15  0858 5E 66 6C 6F 75 7D 85 8A 90 93 99 A1 A6 A9
  30  08B0 B6 BE C4 C7 CD D5 DD E2 E8 EE F3 F6 FD 05
  45  090B 13 1B 21 1E 26 29 29 2F 35 3D 42 45 4C 52
  60  095A 62 64 6C 72 7A 82 8A 90 8D 95 9B A3 A8 AB
  75  09B2 B8 C0 C8 CA CD D3 D8 DE E4 E9 EF F2 F8 FE
  90  0A01 07 0A 0F 15 1B 1E 24 27 2E 34 39 3F 45 4A
 105  0A50 56 5C 5F 62 68 6A 70 73 79 7C 7E 83 89 8E
 120  0A94 96 9B A1 A7 AD AA AD B3 B6 BD C3 C6 CC D2
 135  0AD5 DB DE E4 E7 E9 EF F2 F8 FB FE
  11 FORWARD REFERENCES
P-CODE..  145 INSTRUCTIONS
8080..  766 BYTES
P-CODE:8080 = 1.3206897
* END TRANSLATION *
BYE
*LF PAS.LIB 1A00
*JP0800
??
 29 34 34 35 43 43 235 242
*JP0800
?20
 0 1 5 25 29 29 32 33 34 34 35 35 40 43 43 112 113 201 235 242 244
*JP2A04
READY
!CHR$(129)
```

Listing 1: Compilation and translation of a sample Pascal program. At the end of the translation, the ratio of p-code to 8080 code is determined for reference purposes.

is an immediate BASIC statement used to turn off the printer.)

Summary

Compilers for high level languages are large, nontrivial programs. Their implementation usually requires a significant amount of computer system resources and human effort. Although our available system resources were limited, both in hardware and software, we managed to finish the bootstrap compiler within a relatively short time period. The reason is obvious: The Pascal subset we implemented is small. We followed the same approach professionals use for implementing portable Pascal compilers on mainframe computers. Syntax diagrams, which define the subset language, are used to construct the syntax directed, top-down parser of the compiler. The generation of p-code is also syntax directed. P-code is relocatable and portable, and its interpreter can be easily implemented on most microcomputers.

There are several features that are unique to our compiler project. First, the bootstrap compiler was written in BASIC (North Star disk BASIC). Although BASIC is not an appropriate language for compiler writing, it is the only high level language available in our system. Its ability to perform recursive function calls proved essential in simplifying the implementation of the compiler. Secondly, instead of writing a p-code interpreter in assembly language, a p-code to 8080 machine code translator was written in BASIC. The translated code can be expected to run more than twice as fast as interpreting p-codes. A p-code interpreter with debug facilities was also written (in Pascal). It can be used to debug p-code programs. Thirdly, minor extensions to the subset language were implemented. Absolute addressing of memory locations and machine language interface are desirable features for microcomputer systems. The availability of hexadecimal constants and IO conversions provides much user convenience.

Presently, the bootstrap compiler is very slow. It compiles at the rate of about eight lines per minute for a very dense Pascal program (using North Star BASIC with a 2 MHz 8080 processor). With some refinement in the compiler and runtime routines, the Pascal version of the compiler can be expected to run 25 times faster, or approximately 200 lines per minute.

Completion of the bootstrap compiler is only a milestone in our compiler project. There are many tasks still to be done. Logically the next step is to write the translator and then the p-compiler in the Pascal subset and compile them using the BASIC version of the compiler. Since the compiler source and p-codes are big, there may be a minor problem in memory management. It may be necessary to write the p-codes onto disk to save memory. After these two programs have been debugged, any further development can be done in Pascal without the BASIC interpreter. It would be quite interesting to have the compiler (in object code) compile itself (in source code) and use the output object code to compile itself again. After each compilation, the object code could be compared with the previous one to provide a means of verification.

More Pascal features or extensions can be implemented one step at a time. They may include character type and pointer type variables, disk IO capabilities, floating point arithmetic, multidimensional arrays and built-in functions. It is also necessary to improve the error diagnosis and recovery scheme of the compiler. Further development should be aimed at user convenience. A dynamic debugging package that can display and alter the values of variables as specified by name at runtime would be desirable. Ultimately, we hope to see a Pascal system that is as convenient and easy to use as an interactive BASIC system.■