TR-79-005

A PASCAL CROSS-COMPILER FOR A MICROCOMPUTER

by

Ganlin Jin

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

May 1979

Approved by:

Peter Calingaert

Adviser

Frederick P. Brooks

Reader

Stephen M. Pizer

Reader

GANLIN JIN. A cross-compiler for a microcomputer. (Under the direction of Dr. Peter Calingaert.)

ABSTRACT

This thesis describes an implementation, on a microcomputer, of a cross-compiler for a subset of the Pascal language. The microcomputer chosen here is the MC6800 because of its availability. Two parsing techniques, recursive descent and LR(1), are used in this compiler, which is written in PL/I.

ACKNOWLEDGEMENTS

The author is indebted to Dr. Peter Calingaert for his valuable assistance in the preparation of this thesis, to Dr. K. C. Tai of N.C. State University for his assistance on LR parsing, to Dr. F. P. Brooks, Jr. and Dr. S. F. Pizer for their reading and commenting on my thesis, to Mr. J. E. Leonarz for his administrative help, to many fellow students in this department who assisted me in various aspects, to my wife, Tingmei, for her company.

实践得真知

True knowledge can only be acquired through practice.

TABLE OF CONTENTS

Chapter page

1. INTRODUCTION 1

2. SYNTAX ANALYSIS 3

Language specification 3
Parsing algorithms 4
One-pass compiler 7
Intermediate code 7
Choosing an architecture 7
Choosing the set of I-code 8
I-code optimization 10

3. CODE GENERATION 11

Architecture of 8-bit microprocessors 11
The target machine -- MC6800 12
The architecture of MC6800 12
Constraints imposed by the MC6800 13
Representation of data 15
type integer 15
type byte 16
type real 16
type char and type Boolean 17
Code generation for arithmetic operations 17
Object program loading 18

4. CONCLUSION 20

Extensibility 20
Program testing 20

Appendix page

A. PASCAL-M USER MANUAL 23

5. INTRODUCTION 24

6. THE LANGUAGE PASCAL-M 26

Lexical rules 26
Syntax rules 27

Language differences between standard Pascal and Pascal-M 32
Restrictions 32
extensions 32
Data types 33
Type integer 33
Type byte 33
Type real 33
Type Boolean 34
Type char 34
Scalar and subrange types 35
Array types 35
Standard procedures: input and output 35
Programming examples 36

7. THE IMPLEMENTATION 40

Dimensional limits in Pascal-M 40
Constraints of target machine 41
Job control for running a Pascal-M program 42
Structure of a run 42
Program format 43
Using an object module 43
Input and output 44
Compilation output format 45
Source listing 45
Cross reference and attribute table 45
Trace of compilation 46
Intermediate code 48
Error messages 49
Machine codes 49

B. PROGRAM LOGIC MANUAL 50

8. PROCEDURE STRUCTURE 51

9. LEXICAL ANALYSIS 56

Reserved words 56
Operators 56
Other encodings 57

10. SYNTAX ANALYSIS 58

Encoding of nonterminals 58
Symbol table 59
Declaration part 62
Grammar 62
Error recovery 66
Statement part 67

11. INTERMEDIATE CODE 70

Architecture 70

Specification 72

12. CODE GENERATION 88

Memory organization 88
Run time library 90

BIBLIOGRAPHY 93

Chapter 1

INTRODUCTION

The advent of microprocessors marks the beginning of a new computer revolution in this decade. The widespread application of microprocessors, from process control to small accounting systems, from intelligent terminals to home entertainment sets, indicates the revolution is well under way. And this revolution is far from having run its course.

Since the introduction of the first microprocessors in the early 1970's, microprocessor systems have steadily replaced more and more logic circuits in dedicated control applications. In fact, by far the largest application of microprocessors has been random logic replacement. This trend should continue with the introduction of reasonably powerful one-chip microprocessors.

The programs involved in such applications are much smaller than those for general purpose computers and they are ROM, rather than RAM, based; and more than often, a single copy of a program is replicated thousands of times for certain applications. In the above sense, space efficiency dominates all. Hand-coded machine-language programming is a sure way to achieve that goal; an assembler or a macro assembler should suffice for this purpose. A compiler for a higher level language is not so urgently needed.

On the other hand, nothing in the microprocessor itself implies that it should be used only as logic replacement. With ever increasing complexity and speed and drastic reduction in cost, microprocessor-based systems have already replaced dedicated minicomputers in some cases. Together with memory and peripheral circuitry, processor chips form complete microcomputer systems which are threatening to become truly general-purpose. As a matter of fact, even the dumbest of the microcomputers of today have better performance (speed, reliability, power consumption -- not to mention price) than the 'giant' general-purpose computers of the early 1950's. Microprocessors (or microcomputers) will inevitably retrace the evolution undergone by the 'biggies' in many aspects. For example, people will finally get tired of assembler language programming; will finally feel the importance of software portability; will finally recognize that software costs outweigh hardware costs. The users of the early biggies experienced these same problems before, and the solution to these problems was machine-independent higher-level programming languages, so we might expect that this should be the solution to those problems faced today by the users of microcomputers.

In recognizing these problems and the possible solution to them, people have begun to design and implement higher-level languages for the microcomputers. Now, several languages have been developed for microcomputers, notably PL/M [14], micro-C [11], several dialects of BASIC, and some subsets of Pascal.

This thesis project is to implement a higher-level language on a microcomputer. Motorola MC6800 is chosen as the target machine simply because we have available a microcomputer system, the SwTPC 6800, based on the MC6800 processor.

A subset of Pascal, Pascal-M (described in Appendix A) is chosen as the higher-level language. The advantages of Pascal over other languages are that:

1. Pascal is well known;
2. Pascal is structurally strong [3];
3. Pascal is comparatively easy to implement.

Because of the lack of software support and sufficient memory space in the microcomputer system, it is almost impossible to implement a compiler which runs on the microcomputer. Therefore, the compiler for this thesis project is a cross-compiler -- it compiles Pascal-M source programs on the IBM 370 (under OS/360 MVT Rel.21.8 - HASP II Ver. 3.1) and generates code for the MC6800. The compiler is written in PL/I (OS PL/I Checkout Compiler Ver. I Rel 3.0).

There are logically two phases for implementing any higher-level languages. The first phase is language dependent, extending from lexical analysis through syntax analysis and semantic analysis until intermediate code generation, and probably includes some intermediate code optimization. The second phase is target-machine dependent, including memory management and object code generation, and probably some linking and loading. The next two chapters discuss some problems encountered and describe the thinking behind some of the design and implementation decisions in each of these respective phases.

Chapter 2

SYNTAX ANALYSIS

2.1 LANGUAGE SPECIFICATION

A complete specification of a programming language must perform three functions. First, it must specify the context-free syntax of the language; that is, which strings of symbols are deemed to be well-formed programs. Second, it must specify the semantics of the language; that is, what meaning should be attributed to each syntactically correct program. Third, it must specify the context-sensitive requirements of the language; that is, what are some of the interconnections amoung different segments of a program.

The most commonly used method of syntax specification is by Backus Naur Form (BNF), which has the advantage of being able to specify any context-free grammar, including any ambiguous construct. Another important advantage of BNF is that it can be used as input for automatic parser generators. The disadvantage of BNF is that no semantics is included at all. The use of BNF tends to lead to the intentional or inadvertent introduction of ambiguity where none is present in the language being specified. For example, the famous ambiguity of

```
IF A THEN IF B THEN C ELSE D
```

is caused by specifying the grammar of 'if statement' in BNF as

```
<if stmt> ::= IF <condition> THEN <statement>
  | IF <condition> THEN <statement> ELSE <statement>
<statement> ::= <if statement> | <other statement>
```

This is easily resolved by letting shift action dominate reduce action whenever a such conflict occurs. Another example is about parameter passing in Pascal. In passing parameters to subroutines, either call by value or call by reference could be used, depending on how formal parameters are declared. At the calling point, the reduction from expression or variable to actual parameter is specified in the Pascal -- User Manual and Report [15] (subsequently referred to as the Report) as

```
<actual parameter> ::=  <expression>
                      | <variable>
```

Together with the commonly used reduction

<expression> ::= <variable>

this forms an ambiguous construct. The ambiguity is caused by specifying a context-sensitive construct by a context-free grammar.

Another commonly used method of syntax specification is by syntax diagram, which has the advantage of being able to let people grasp an intuitive feeling about the grammar easily, just as with statistical diagrams, i.e., graphs (as opposed to statistical tables). It is very helpful in directing people to write recursive descent parsers. The major disadvantage of syntax diagrams, besides their saying nothing about semantics, is that it is not possible to process them mechanically; thus they cannot be used as input for automatic parser generators. Another important disadvantage of syntax diagrams is that it is not always possible to represent a given programming language by means of syntax diagrams. For example, the syntax diagram specification of the previous example on actual parameter in the Report is:

Actual parameter

$$\longrightarrow \boxed{\text{expression}} \longrightarrow$$

which has only the first half of the BNF equivalent.

Specification of the semantics and context-sensitive requirements of a language is usually done by words, though there are some formal definitions available [16]. The syntax specification of Pascal-M (appendix A) is done both in BNF and in syntax diagrams in a complementary way, as is done in the Report. The semantics and context-sensitive requirements of Pascal-M are the same as specified (in words) in the Report.

2.2 PARSING ALGORITHMS

The parsing algorithms used in this compiler were chosen from the many standard parsing algorithms commonly available [1,2,8,10,13]. In general, the standard parsing algorithms can be classified into two categories: top-down and bottom-up. The terms refer to the way the syntax tree is built. A representative top-down parsing algorithm is recursive descent [13, pp. 97-100], which has the following advantages. It is straightforward to understand, the parser is easy to write, all parsing actions are well under human control, and no backtracking is necessary. But it requires a language that allows recursive calls to implement the parser.

Bottom-up parsing culminates in LALR(1) (one symbol look-ahead left-to-right scan rightmost derivation) [2] which is the most efficient of all parsing algorithms, and the parser can be generated automatically. Theoretically the languages accepted by LALR(1) (or loosely LR(1)) are a subset of *unambiguous* context-free languages. Actually, LR parsers can be generated for ambiguous grammars too. And the intentional rewriting of an unambiguous construct into an ambiguous one can even be exploited to reduce the number of nonterminals and thus the number of productions [1, pp. 116-119]. The secret lies in an important feature of the parser generation algorithm; that is, the automatic detection of ambiguities and difficult-to-parse constructs in the language specification. The pitfalls detected can be used to guide human compiler writers to modify the output parser according to their knowledge. For example, in this Pascal-M thesis project, <expression> is written for the parser generator input as

```
<expression> ::=
     <expression> <relational operator> <expression>
   | <expression> <adding operator> <expression>
   | <expression> <multiplying operator> <expression>
   | <sign> <expression>
   | ( <expression> )
   | <variable>
   | <unsigned constant>
   | not <expression>
```

instead of the Report specification

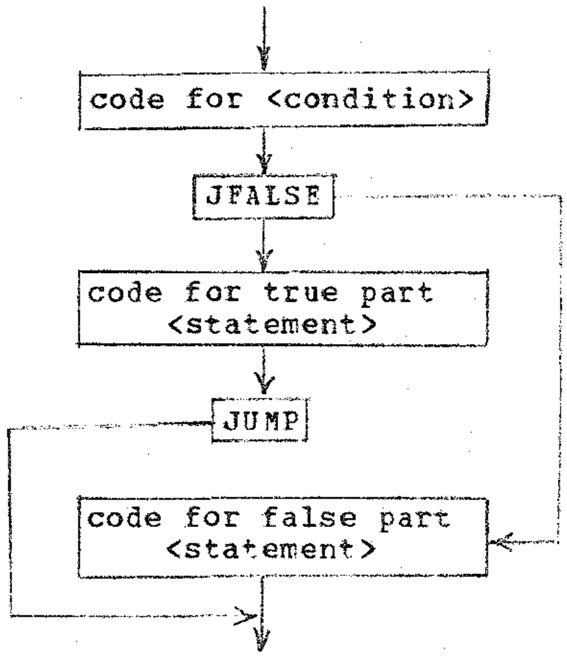
```
<expression> ::= <simple expression>
   | <simple expression> <relational operator>
             <simple expression>
<simple expression> ::= <term>
   | <sign> <term>
   | <simple expression> <adding operator> <term>
<term> ::= <factor>
   | <term> <multiplying operator> <factor>
<factor> ::= <variable>
   | ( <expression> )
   | <unsigned constant>
   | not <factor>
```

Because the former has fewer nonterminals, it is more efficient.

Though LR parsers have so many advantages, LR parsing is not a panacea. First, a parser generator must be available. Second, parsing actions are difficult for human to comprehend; should anything go wrong, it is hard to debug. Third, the grammar of the language must be rewritten here and there to match the nature of LR parsing. During LR parsing there are two parsing actions: *shift* and *reduce* [1]. Shift actions do nothing more than stacking a new state and shifting to a new token. Only during reduce actions are semantic routines called into play. Difficulties arise if some semantic routines should be called at the point where only shift action is taken. For example, if the <if statement> is specified in BNF as

<if statement> ::=
    IF <condition> THEN <statement> ELSE <statement>

the code generated from <if statement> could be characterized by the following diagram.



When the LR parser encounters THEN or ELSE, it will take shift action; thus the JFALSE, JUMP and patching of their destinations will not be generated. If the grammar is rewritten as

<if statement> ::= IF <condition> <then> <statement>
                    <else> <statement>
<then> ::= THEN
<else> ::= ELSE

then we can generate code for the two jumps and patches during the reduction to <then> and <else>. A small negligence will cause the whole parser generation program to be rerun, which is often costly. Therefore, the grammar must be carefully examined and modified before processing by the parser generator.

Beside checking the validity of program syntax, two other functions are incoporated into the parser, building the symbol table, and driving semantic routines. Therefore, it is more convenient to divide a program into two parts, a declaration part and a statement part, which correspond to the two functions.

In this thesis project, recursive descent is used to parse the declaration part and LR(1) is used to parse the statement part. (The parser generator used in this thesis project is actually SLR(1) [9], which is almost the same as LALR(1); the only difference, if any, is that the LALR(1) might have smaller look-ahead sets.)

The advantages of LR(1) parsing and the current trend toward using this parsing method encouraged me to use it, but initial lack of confidence with the LR(1) parser generator and technique made me decide to perform only part of the syntax analysis with it. Since recursive descent is a top-down method, any portion of the language can be parsed by a different method, and the two methods mesh naturally.

2.3 ONE-PASS COMPILER

The Pascal language is designed to be implementable by *one-pass* compilers. This means that each input string in the source program is read in only once, and the parser will never have to go back from the beginning in order to decide what actions to take. One-pass compilation implies efficiency, but not all languages are implementable in one pass. To make a language one-pass implementable, certain restrictions must be imposed. In Pascal, for example, the most striking and unpleasant feature is that all *goto* labels must be explicitly declared. Another feature is that 35 key words are reserved, so that their attributes are fixed before parsing. A third Pascal feature, though advertised as 'discipline of programming', is also a consequence of permitting one-pass compilation: all variables must be explicitly declared.

Since Pascal-M is a subset of Pascal, all the restrictions of Pascal caused by the one-pass assumption also hold for Pascal-M. It is natural to take advantage of this and construct a one-pass compiler.

2.4 INTERMEDIATE CODE

2.4.1 Choosing an architecture

Many different representations of intermediate code exist. The most common are: postfix, quadruples, triples, and indirect triples.

Postfix notation or Polish notation [13, pp. 247-252] is particularly attractive for the computer representation of arithmetic expressions. Explicit naming of intermediate results is not necessary because an operand stack is used. The major disadvantage of postfix notation is that it is not instruction-like, being a continuous flow of a mixture of operators and operands. The representation of each entry in this flow must be able to accommodate the largest of all possible operators and operands. Besides, this continuous flow without pause is hard for humans to follow.

Quadruples (operation code, first operand, second operand, result) [13, pp. 252-254] representation is instruction-like, with distinct fields for operators and for operands; it remedied the disadvantage of postfix notation. But a lot of temporary variables are introduced into quadruples, constituting a major disadvantage.

The spell of temporary variables that haunted quadruples is broken by the triples [13, pp. 254-256] representation, which saves about a quarter of the space by having one less field (the result field) than the quadruples. But a level of indirection is introduced instead. Indirect triples [13, pp. 256-257] offers further savings in space, but introduce yet another level of indirection.

Another representation, which has been chosen for this thesis project, is a variant of the so called P-code [4,22], which is instruction-like and assumes a hypothetical stack machine, similar to the Burroughs B5000 [18], as its target machine. Because individual entries are instructions, P-code has the advantages of both postfix notation and quadruples. The hypothetical stack machine of this thesis project is based on the PL/0 processor [22, pp. 331-333], with the stack modified to being only one byte wide and with some instructions added. The details of the intermediate code used are described in Appendix B.

2.4.2 Choosing the set of I-code

After deciding the form of I-code (intermediate code), the next thing is to choose the specific representation. Beside consistency, there are three more or less conflicting criteria in deciding what kind of operations should be included.

1. Convenience criterion. This seeks convenience for the semantic routine to use. For example, in Pascal-M there are six relational operators (>, >=, <, <= ,=, ¬=). If we have all six corresponding I-code operations (e.g. GT, GE, LT, LE, EQU, NEQ),

it will be most convenient for semantic routines. Because the operands for the relational operators could be of different data types, it might be convenient to have corresponding mixed operations, too (e.g. for the '>' operator, we might have GTB, GTBI, GTIB, GTI, GTR... etc., where B, I, R indicate byte, integer, and real operands, respectively). The convenience criterion tends to lead to proliferation of operations.

2. Parsimony criterion. The bigger the I-code operation set, the more code generation routines are needed. The parsimony criterion tends to lead to a smaller I-code operation set, and a minimum sufficient set is desirable under this criterion. The minimum set is similar to the basis vectors for an n-dimensional space in linear algebra, in that any operations in this set are linearly independent of each other. For example, the minimum set of the above six relational operators could be four: GT, GE, EQU, and a COM (complement). The other three could be formed by combining two operations in the basis set. Besides possessing independence, the operations in the basis set should be in some sense orthogonal, so that all other operations could be formed by a shorter combination of the basis set. Parsimony often leads to inconvenience, and the I-code program tends to be longer than had this criterion not been honored.

3. Efficiency criterion. Under the efficiency criterion, the object code generated from the I-code program must be short in size and fast to execute. Object code efficiency can be achieved by providing many specialized I-code operations; it again, like the convenience criterion, will lead to proliferation of operations.

Besides the three foregoing criteria, the expected extent of code optimization will influence choosing the set. For example, if we decide to include only four operations for the six relational operators according to the parsimony criterion, will that lead to longer code because '<' will be translated into GE and COM rather than only LT? The answer is no, because the relational operations in Pascal-M will appear only as tests for conditions, any relational operation will always be followed by a conditional jump, and a simple optimization program later can change the pair COM and JPF (jump if false) into JPT (jump if true), or from COM and JPT into JPF, so there will be no significant loss in not providing the three operations LE, LT, and NEQ.

2.5 I-CODE OPTIMIZATION

Code optimization is usually a non-trivial task. Since code optimization is not the primary goal of this thesis project (The primary goal is a compiler that works.), only the following simple I-code optimization are included in this compiler.

1. Type byte constant folding.

2. Load-store pair cancellation.

3. Indirect jump elimination.

4. COM-JPF, and COM-JPT pair transformation.

Chapter 3

CODE GENERATION

3.1 ARCHITECTURE OF 8-BIT MICROPROCESSORS

The target machine of the compiler which was constructed could be any 8-bit microprocessor, simply because there are so many similarities among them. Though this compiler currently generates code only for the MC6800, it will be more instructive to understand first the architecture of 8-bit microprocessors in general in order to visualize the problems associated with code generation for this particular class of target machines.

The following is a summary of 8-bit microprocessor architecture characteristics [7].

1. Short word size: 8 bits only.

2. Short operation code, usually 8 bits.

3. Variable instruction length -- which saves memory space.

4. Many address abbreviation techniques:

   a) implicit operand.

   b) immediate operand.

   c) relative branch.

   d) indexed or based addressing.

   e) register-to-register operations. Often the registers are implicitly specified in the operation code rather than explicitly in the operand address field.

   f) many addressing modes.

5. Few accumulators, few index registers.

6. Stack for subroutine linkage and interrupt handling.

7. No mutiplication, no division, nor anything that requires subsequencing.

All the above characteristics reflect the fact that space efficiency dominates all in the realm of microprocessors.

3.2 THE TARGET MACHINE -- MC6800

3.2.1 The architecture of MC6800

The following is a summary of MC6800 architecture:

1. Programmable registers:
   Accumulator A ----- 8 bits
   Accumulator B ----- 8 bits
   Index register --- 16 bits
   Stack pointer ---- 16 bits
   Program Counter --- 8 bits
   Status Register --- 8 bits

2. Two's complement number representation.

3. Memory addressing modes:

   a) Immediate addressing. Instructions of this group are 2 bytes long; the second byte is the operand.

   b) Direct addressing. Instructions of this group are 2 bytes long; the second byte specifies the address of an operand located in the first 256 bytes of memory address space.

   c) Extended addressing. Instructions of this group consist of 3 bytes; the second and third bytes form a 16-bit operand address. This addressing mode has the ability to access the full range of the memory space (65,536 bytes).

   d) Indexed addressing. Instructions of this group consist of 2 bytes; the second byte is an offset which will be added to the content of the index register and the 16-bit sum will be the operand address.

   e) Inherent addressing. Instructions of this group have only one byte; the operand(s) are implicitly specified by the operation code.

4. Branch instructions. There are four addressing modes in branch instructions: relative, indexed,

extended, and inherent. In the relative mode of branching the second byte is the offset from the current program counter value. The offset has a range of [-128, +127].

5. Status flags. The status register stores six flags: carry, overflow, sign, zero, half carry, and interrupt mask respectively. Only half carry is unusual. This flag will be set whenever a carry from bit 3 to bit 4 occurs on the last operation. It is included in order to facilitate decimal operations. The leftmost two bits of the status register are not used.

6. I/O and memory are within a single address space. Thus all I/O devices are addressed as memory locations.

3.2.2 Constraints imposed by the MC6800

All user programmable operations are performed on 8-bit data in the MC6800. Any operation that requires more than eight bits must resort to software multiple-precision routines. Though the architecture of the MC6800 has provisions for implementing multi-precision operations (e.g. the carry condition and all operations that involve it), any such attempt will be painfully slow. To make matters worse, the MC6800 has only one index register and no other indirect addressing facilities. This demands that all indirect addressing be implemented through the lone index register. This is a heavy blow on 'block structured' languages, since block structure requires activation record memory management, and all accesses to variables are through the activation record indirectly. Besides activation record management, array indexing and arithmetic operations on the operand stack all require indirect addressing. The lone index register must be loaded and stored frequently to shuttle among different uses, which wastes a lot of time. Another disadvantage of having only one index register is that it is impossible to access efficiently two data structures that are more than 256 bytes apart. The reason is that though the index register is 2 bytes long, the offset has only one byte.

As an example of the last point, the following is a real problem that I encountered in a head motion parallax project. The main idea of that project is to use photo-sensitive devices to find the position of the head of a human observer [12]. Finding the head position in one dimension can be reduced to an edge finding problem. For a 1728-element sensor, the program looks like the following:

```
FOR I:=1 TO 1728 DO
 BEGIN
  (* SUBTRACT DARK LEVEL *)
  A(I):=A(I)-B(I);
  (* FILTER  NOISES  AND  USE   *)
  (* LAPLACIAN TO FIND THE EDGE *)
  . . . . . . .
 END;
```

The line A(I):=A(I)-B(I) seems quite simple, but we know that the arrays are stored at least 1728 bytes apart. The offset of indexing is not able to distinguish the two data structures if a single index value is maintained; we must resort to loading and storing the index register twice per iteration. To simplify the illustration of this point, we ignore the problems of activation record management (which needs additional indirection), and assume element types of both A and B are type byte. The translated code for A(I):=A(I)-B(I) would look like:

```
LDX  BIX       Index reg := BIX
LDAB 0(X)      AccB := B(I)
INX            Index reg := Index reg + 1
STX  BIX       BIX := Index reg
LDX  AIX       Index reg := AIX
LDAA 0(X)      AccA := A(I)
SBA            AccA := AccA - AccB (A(I) - B(I))
STAA 0(X)      A(I) := AccA
INX            Index reg := Index reg + 1
STX  AIX       AIX := Index reg
```

In constrast, if the first line of the above program were
FOR I:= 1 TO 100

and both A and B were declared as arrays of dimension 100, then the translated code for A(I):=A(I)-B(I) would be:

```
LDAA 0(X)      AccA := A(I)
SUBA 100(X)    AccA := AccA - B(I)
STAA 0(X)      A(I) := AccA
INX            Index reg := Index reg + 1
```

What a difference!

The Motorola company has finally realized these problems, too. Their recent announcement [19] on their next generation 8-bit microprocessor, the MC6809, showed the following improvements, all of which will contribute to easing the constraints of the MC6800 and facilitate higher-level language programming.

1. another index register.
2. another user data stack, beside the linkage stack.
3. 16-bit offset indexing, beside the old 8-bit offset.

4. a direct page base register.

3.3 REPRESENTATION OF DATA

3.3.1 Type integer

For most languages, type integer is the basic and most important data type. It is used in representing integer numbers used as arithmetic operands, as indices for loop control, as subscripts for array accessing, or as codes in encoding some information which does not necessarily have any direct relation with integers at all. In some tiny languages or some inexpensive implementation of certain bigger languages, only type integer is provided to the user, and the users are forced to encode other data types by means of integers. On the other hand, in some big languages, like PL/I or ALGOL-68, in addition to type integer and other data types, half, quarter and various multiple-precision integers and integer representations on different basis are provided. In such cases, often even the name 'integer' is subdivided (e.g. in PL/I, FIXED BINARY, FIXED DECIMAL etc.).

What is an integer then? If we use a formal mathematical definition, it would be impossible to use a fixed number of bits to represent all possible integers. Therefore, a practical approach is used in the Report (p. 13): a value of type integer is an element of the implementation defined subset of whole numbers.

For bigger machines, the choice of how to represent an integer is simple: a 'word' is a natural candidate, and the compiler designer can simply use whatever representation for a word is specified by the machine architecture, be it 1's or 2's complement, signed magnitude, or whatever. This will simplify the implementation of operations on integers.

For 8-bit microprocessors the word size is eight bits, which is often not sufficient to represent the needed subset of integers. (Actually eight-bit word size is more than sufficient for a lot of control applications.) The concepts of word and integer must be separated, for we cannot use a word or byte to represent all of the integers we need. Two bytes concatenated together could more often satisfy our needs, but it would make operations on integers complicated. Therefore I decided on using two representations: two bytes for type integer, and one byte for <u>type byte</u> (short integer), which is an extension to standard Pascal data types. The standard type integer will be sufficient to represent the most commonly used whole numbers. Type byte possesses, however, the property of having the most efficient implementation. Both types are represented in 2's complement form. Type integer has the range [-32768, +32767]; type byte has the range [-128, +127].

3.3.2 Type byte

As described in section 3.3.1, type byte is intended for efficient implementation of short integers. In order to make full utilization of this efficiency, not only variables but literals should be represented in one byte whenever possible. For example, in the following declaration:

```
CONST A=10;
      B=100;
      C=1000;
```

constants A and B could and should be translated into type byte, constant C into type integer. It would be very inefficient to treat small literal numbers as type integer, as for the '1' in N:=N+1. Sometimes mixed type operation or automatic type conversion is required. If we have this facility, then the literal '1' would be translated into one byte, independently of the type of N.

3.3.3 Type real

The choice for representing real numbers (or floating-point numbers) on microprocessors is not an easy task. The most common practice for big machines is to use distinct representations for exponent and mantissa: biased representation for exponent and signed magnitude representation for mantissa.

For the microprocessors, the architecture specifications have no floating point at all. The choice of representation should be based on the overall efficiency of implementing all arithmetic operations, including normalization and conversion. The evaluation of efficiency for these operations, which are actually software subroutines, should be based on the machine language level, not on the microprogramming or hardware level (see section 4.4).

After some study, I decided on 3-byte precision: one byte for the exponent and two bytes for the mantissa. The harder decision, i.e., how to represent them, was narrowed to the following two choices.

1. Use a uniform representation, i.e., represent both exponent and mantissa by 2's complement.

2. Use distinct representations, e.g. those of IBM 360.

Either choice has some advantages over the other. For example, under the first choice, multiplication is straightforward to implement; just add the two exponents and multiply the two mantissae. The second choice is selected for this thesis project, however, for the following reasons.

1. It offers a greater degree of compatibility with big machines.

2. It allows the use of fixed-point instructions for comparing the magnitude of floating-point numbers.

The base for the exponent is 2 in this thesis project, instead of 16 (as in the IBM 360), because that will cause less precision loss during normalization; however, the range is shortened to $|10^{-16}, 10^{+16}|$, instead of about $|10^{-76}, 10^{+76}|$.

3.3.4 Type char and type Boolean

A variable of type char is naturally represented as one byte. A variable of type Boolean is also represented as one byte in this project. This is not a sacrifice of space for time. For the MC6800, the address resolution is to the byte. Should we represent a Boolean variable by one bit, then any manipulation on it would require more bytes of programming effort, which would waste much more space than it saved.

3.4 CODE GENERATION FOR ARITHMETIC CPERATIONS

Because microprocessors have no multiplication or division operations, and no multiple-precision nor floating-pointing operations, all these operations must be programmed. Since it requires dozens of bytes of coding for each of these routines, it is most convenient to store them in a library and call them whenever such an operation is encountered rather than generate the whole segment of code for each occurrence. It wastes time to perform subroutine linkage for each occurrence of those seemingly simple operations, but there is no alternative under such target machine architecture. I considered the use of 'threaded code' [4], but it turned out to be of no use for this one index register machine.

The best we can do is to program these routines as efficiently as possible. There are a lot of algorithms for performing multiplication and division [6], but those algorithms are for coding on the microprogram level, not on the machine-language level. We must re-evaluate the efficiency of each algorithm from the level of machine language. For example, the 'one multiply' should be twice as fast as the 'simple shift multiply', according to [6]. In [17, p. 2.15], which uses one multiply to do double-precision multiplication (16 by 16 bits, with 32-bit result), 78 bytes of coding are used, with an average time of about 1180 cycles. I used simple shift multiply to implement the same operation, which cost me 45 bytes and an average time of about 971 cycles. Does this contradict the theory in [6]? No, because logical operations at the microprogram level are much faster than addition, whose speed is limited by carry propagation time. Thus those algorithms (one multiply, two multiply, etc.) in [6] are trying their best to minimize additions by using more logical operations. That condition is not true at the machine-language level -- logical operations are of the same speed as addition or subtraction. Thus in order to reduce one addition by using several more logical operations is not justified at the machine language level, contrary to the examples in [17]. The rule of thumb for a good algorithm for a machine-language routine is: the fewer bytes of coding the better.

3.5 OBJECT PROGRAM LOADING

The object code generated by this cross-compiler must be loaded into the RAM (random access memory) of the microcomputer. This section presents a means for doing this.

There are two ways to enter data into our microcomputer system:

1. Through ACIA's (asynchronous communications interface adapter). We have two of them: one is connected to a HP 2645 CRT terminal (the console for the system), the other is connected to a floppy disk system.

2. Through PIA's (parallel interface adapter). We have only one PIA which is configured as output only and is currently connected to the PDP-11/45 for the head motion parallax project.

There is no direct connection between the source machine (IBM 370) and the target machine (MC6800). Upon first glance at such a system configuration, the only way to load the object code into the microcomputer appears to be by manually typing in the object code through the console terminal. Alternatively we are forced to change the system configuration. (For example, by replacing the CRT terminal with a clumsy teletype, we could load through paper tape.)

Fortunately, the HP 2645 is an intelligent terminal with a screen buffer of 2249 bytes. This suggests the following solution.

step 1: Connect the HP 2645 to the IBM 370.

step 2: Display the object code on the screen of the HP 2645.

step 3: Switch the connection of the HP 2645 from IBM 370 to the microcomputer.

step 4: Load the object code from the screen buffer into the RAM of the microcomputer.

Chapter 4

CONCLUSION

4.1 EXTENSIBILITY

This compiler is built with future extension in mind; therefore, the following decisions were made:

1. The syntax analysis is for the whole standard Pascal.

2. All unimplemented language features will be caught and error messages printed for them, but the parser will keep on going normally. The appearance of an unimplemented feature will not cause the parser to collapse nor to generate a lot of spurious error messages.

3. The unimplemented features of the declaration part will not be entered into the symbol table.

4. The unimplemented features of the statement part will have null reduction (semantic) routines.

5. For some I-code instructions, code generation is not implemented.

Future inclusion of a certain unimplemented feature will only have to: 1) enter some additional information into the symbol table, or 2) fill up an empty semantic routine, or 3) add a subroutine into the operations library for the target machine. The skeleton of the whole compiler will not be touched at all.

4.2 PROGRAM TESTING

It is hard to prove the correctness of even a moderate program, not to mention such a compiler (which has about 6000 lines of PL/I source program). To test the compiler, I chose some representative Pascal-M programs as test data, compiled them on an IBM 370 and loaded and ran the object programs on the MC6800 microcomputer. The following three programs were compiled and run correctly, as well as all the example programs in Appendix A. Therefore, at least the features involved in those programs could be considered dependable.

1. A greatest common divisor program using Euclid's method, a complete simple program.

2. A factorial program, a simple example of recursive programs. Its correct running proved the success of activation record storage management and parameter passing mechanisms (both call by value and call by reference).

3. A quicksort program (QSORT) shown following. This is a more complex program, which includes a lot of features: _if_ statement, _for_ statement, _repeat_ statement, _while_ statement, _goto_ statement, recursive program, local and global variables, _array_ variable, user defined type, and input and output procedures.

At this writing, the following features of Pascal-M are either not fully implemented or not tested completely: multi-dimensional arrays, real number operations, and character operations.

Some statistics about the compilation of the QSORT program might be interesting. The intermediate code generated by the semantic routines has 197 instructions. After optimization, only 134 instructions are left. The generated object code for the 134 instructions takes 802 bytes (not including the run-time library).

```
PROGRAM QSORT;
 TYPE LIST=ARRAY[ 1..30 ] OF BYTE;
 VAR N:BYTE; (* # OF ELEMENTS TO BE SORTED *)
     A:LIST;
     I:BYTE;
 PROCEDURE QUICKSORT(L:BYTE;R:BYTE);
  LABEL 100;
  VAR I,J,K,T:BYTE;
  BEGIN
    IF L<R THEN
     BEGIN
       I:=L; J:=R+1; K:=A[L];
       WHILE TRUE DO
        BEGIN
          REPEAT I:=I+1 UNTIL A[I]>=K;
          REPEAT J:=J-1 UNTIL A[J]<=K;
          IF I<J THEN BEGIN
                        T:=A[I];
                        A[I]:=A[J];
                        A[J]:=T
                      END
                 ELSE GOTO 100;
        END;
       100: T:=A[L];
            A[L]:=A[J];
            A[J]:=T;
       QUICKSORT(L,J-1);
       QUICKSORT(J+1,R)
     END
  END;

 BEGIN (* MAIN PROGRAM *)
   READB(N);
   FOR I:=1 TO N DO READB(A[I]);
   A[N+1]:=127;
   QUICKSORT(1,N);
   FOR I:=1 TO N DO WRITEB(A[I])
 END.
```

Appendix A

PASCAL-M USER MANUAL

Chapter 5

INTRODUCTION

Pascal is a language designed by Niklaus Wirth to be easily and efficiently implementable on big computers, while at the same time being a suitable vehicle for teaching programming in a systematic and well-structured fashion. *Pascal-M* is a dialect of Pascal designed for 8-bit microprocessors. This manual describes a cross-compiler for Pascal-M written in PL/I. Chapter 2 of this manual defines the language Pascal-M relative to standard Pascal. Chapter 7 describes current implementation of Pascal-M. We will use 'Pascal, user manual and report', 2nd edition by Jensen and Wirth [15] (we will simply call it the Report throughout this manual) as the definition for standard Pascal.

This manual however, assuming the user has a reasonable acquaintance with standard Pascal, will not attempt to teach the user how to program in Pascal. It will only describe the implementation-dependent features and deviations from standard Pascal. For users who are not familiar with Pascal, we recommend [9,15,22].

For programmers acquainted with ALGOL, PL/I, or FORTRAN, it may prove helpful to glance at Pascal in terms of these other languages. For this purpose, we list the following characteristics of Pascal (which also hold for Pascal-M):

1. Declaration of variables is mandatory.

2. 35 key words (e.g. *begin*, *end*, *while*, etc.) are reserved and cannot be used as identifiers. In this manual they are underscored. (Depending on context, underscoring is also used to emphasize certain key phrases in this manual.)

3. The semicolon (;) is considered as a statement separator, not as a statement terminator (as it is in PL/I).

4. Besides standard data types, there is a facility to declare new, basic data types with symbolic constants.

5. Arrays may be of arbitrary dimension with arbitrary bounds; the array bounds are constant. (i.e. there are no dynamic arrays.)

6. As in FORTRAN, ALGOL, and PL/I, there is a goto statement. Labels are unsigned integers and must be declared.

7. The compound statement is that of ALGOL, and corresponds to the DO group in PL/I.

8. The facilities of the ALGOL switch and the computed goto of FORTRAN are represented by the case statement.

9. The for statement, corresponding to the DO loop of FORTRAN, may only have steps of 1 (to) or -1 (downto) and is executed only as long as the value of the control variable lies within the limits. Consequently, the controlled statement may not be executed at all.

10. There are no conditional expressions and no multiple assignments.

11. Procedures may be called recursively.

12. There is no 'own' attribute for variables (as in ALGOL).

13. Parameters are called either by value or by reference; there is no call by name.

14. The 'block structure' differs from that of ALGOL and PL/I insofar as there are no anonymous blocks, i.e. each block is given a name, and thereby is made into a procedure.

15. All objects -- constants, variables, etc. -- must be declared before they are referenced.

Chapter 6

THE LANGUAGE PASCAL-M

6.1 LEXICAL RULES

The lexical rules are essentially those specified in the Report. In deference to the EBCDIC character set, however, a few lexical substitutions must be made:

Report	Pascal-M
{ }	(* *)
arrow	@

(though the pointer is not implemented in Pascal-M, it is included for future extension)

Additionally, some symbols may be entered as shown in the Report, or they may be abbreviated.

Report	Pascal-M
<>	¬=
and	&
or	|
not	¬

The underscore character is accepted as a letter, too. To accommodate EBCDIC as well as ASCII terminals, Pascal-M accepts : and ! in place of [and], respectively.

The 35 reserved words are the same as in the Report. They are listed here for easy reference.

and	downto	if	or	then
array	else	in	packed	to
begin	end	label	procedur	type
case	file	mod	program	until
const	for	nil	record	var
div	function	not	repeat	while
do	goto	of	set	with

The reserved words *file*, *in*, *nil*, *packed*, *record*, *set*, and *with* are not included in current Pascal-M, but they are reserved nevertheless for the sake of compatibility with standard Pascal and provision for future extension.

In addition to the 35 reserved words, there are 13 words have predefined meaning in Pascal-M. But unlike reserved words, these words can be redefined by the user. The following are the 13 words and the class each of them belongs to.

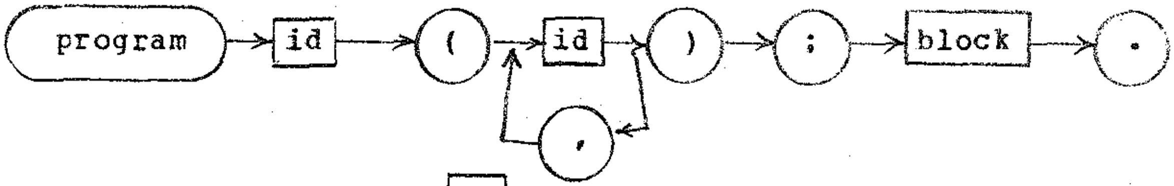
predefined words	class
Boolean	type id
byte	type id
integer	type id
char	type id
false	Boolean constant
readb	standard procedure
readi	standard procedure
readr	standard procedure
real	type id
true	Boolean constant
writeb	standard procedure
writei	standard procedure
writer	standard procedure

All identifiers are recognized by their first 8 characters. If they are longer than 8, the rest will be ignored, as suggested by the Report.

In a Pascal-M program, lower case letters are not allowed as symbols.

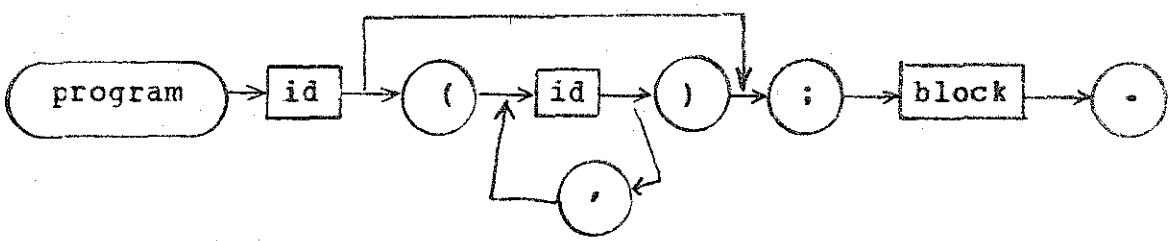
6.2 SYNTAX RULES

For the sake of compatibility with standard Pascal and provision for future extension, the syntax rules of Pascal-M are made almost the same as those specified in the Report. The only exception is that of *program heading*. The syntax graph of a program as specified in the Report is:



where the id between the parentheses denotes file identifier(s).

Since external files are not fully exploited in Pascal-M, those file identifiers could be omitted. Thus, the syntax graph of this part is modified to:

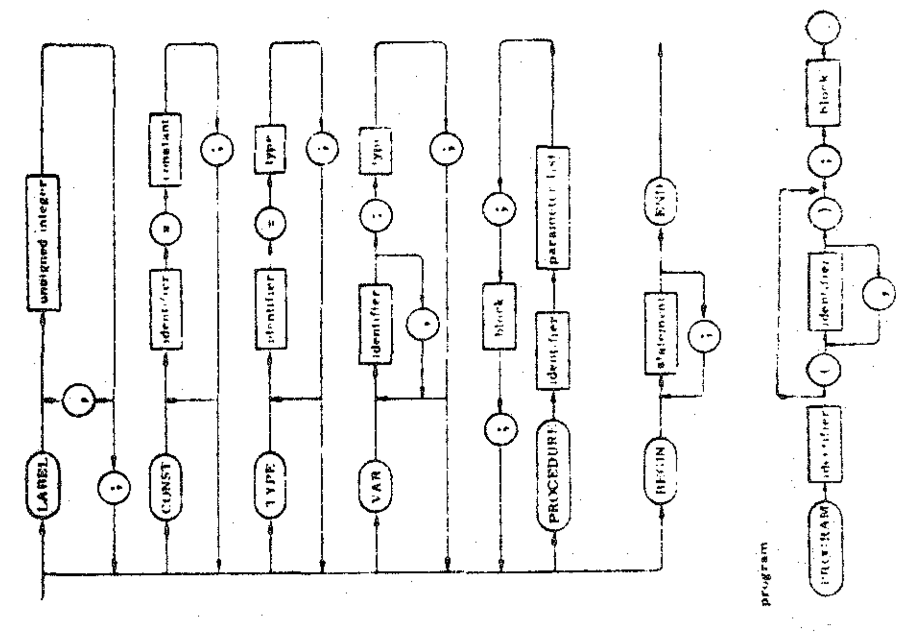


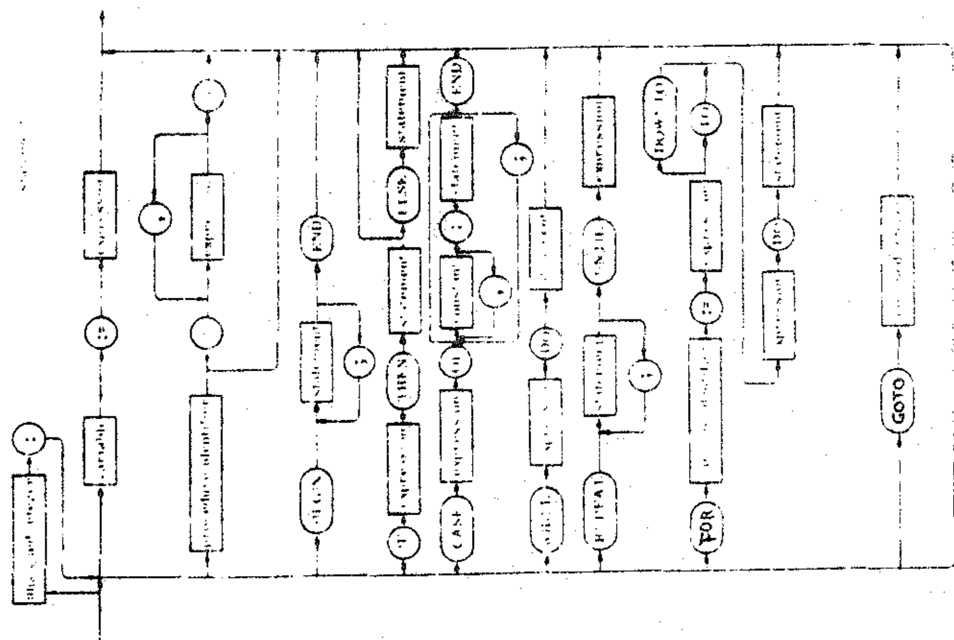
If any file identifier exists, it will be ignored.

Although some features of standard Pascal are not included in Pascal-M (e.g. <u>record</u> type), the syntax analysis part of the compiler will process the whole language, if it is within the dimensional limits of section 7.1. All attempts to use unimplemented features will be caught in syntax analysis and error messages will be printed, and generation of code will be suppressed; they will not cause the compiler to collapse, however. The actual semantically meaningful syntax of Pascal-M is printed on the following pages in syntax graph form for easy reference. Those graphs are copied from the Report (pp. 116-118), with all unimplemented features deleted.

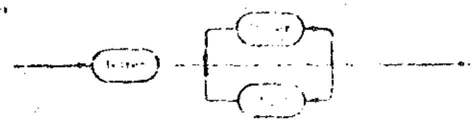
block

program



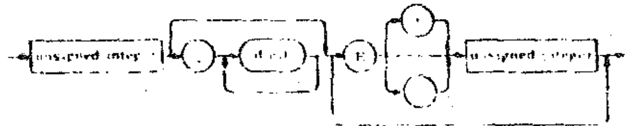


identifier

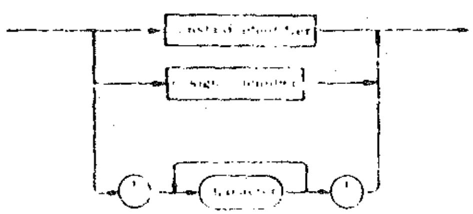


unsigned integer

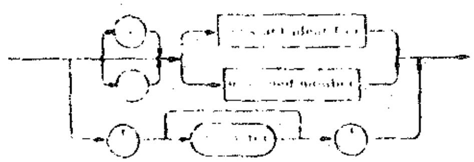
unsigned number



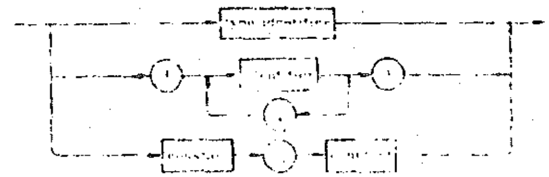
unsigned constant



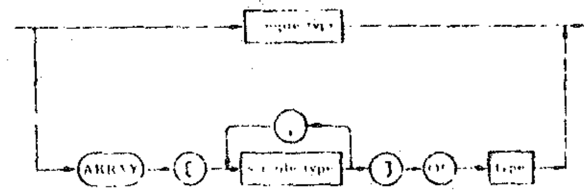
constant

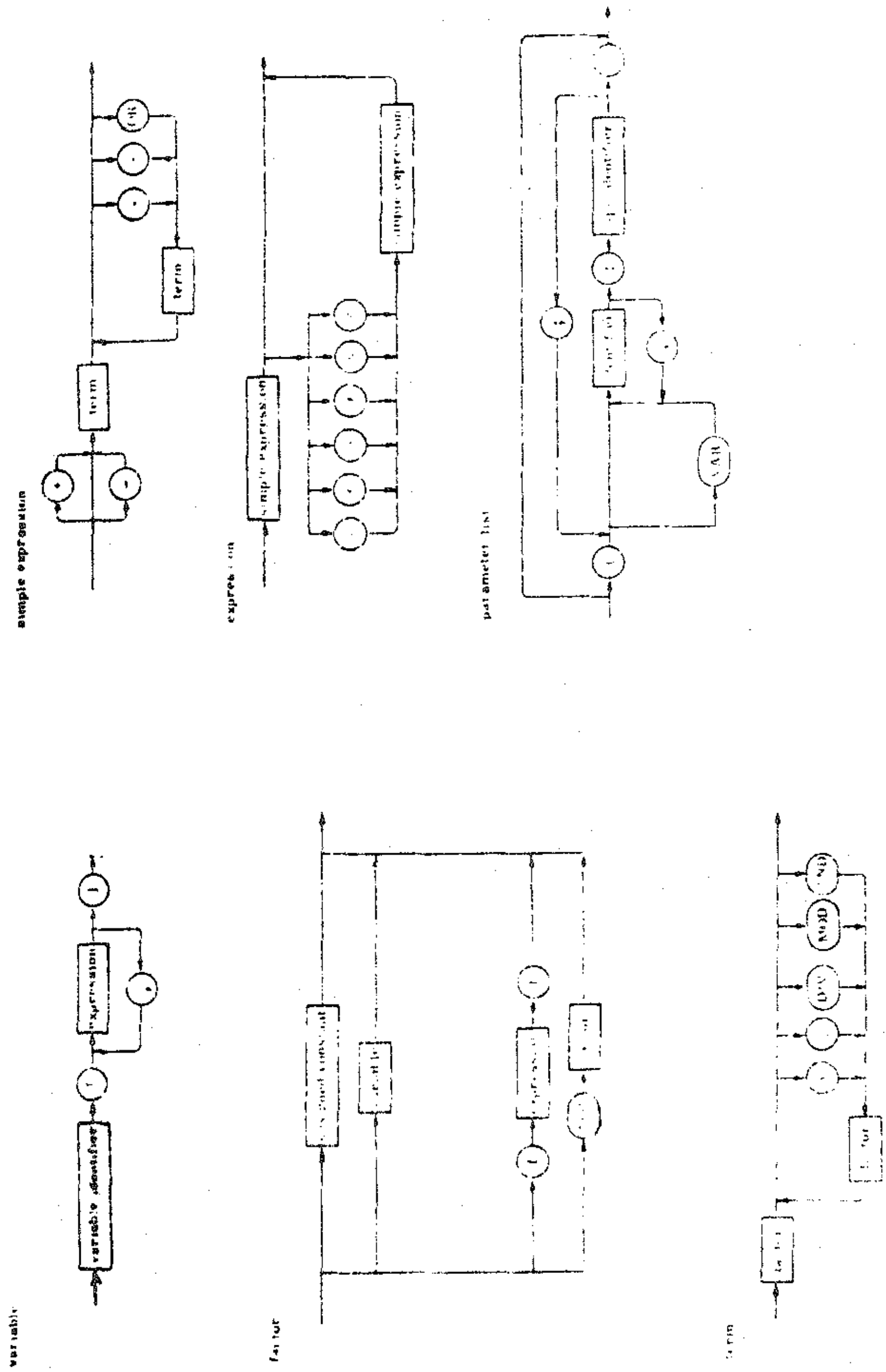


simple type



type




simple expression
term
OR
expression
simple expression
parameter list
VAR
variable
variable identifier
expression
factor
unsigned constant
term
factor
DIV
MOD
AND

6.3 LANGUAGE DIFFERENCES BETWEEN STANDARD PASCAL AND PASCAL-M

6.3.1 Restrictions

Since this compiler is a relatively small-scale project, only a subset of standard Pascal language is included in Pascal-M. The following are the major restrictions of Pascal-M. (Some points can be seen from the syntax graphs of section 6.2.)

1. There are no record, set, file, and pointer types.

2. There is no function facility, and hence no standard functions.

3. Procedure names cannot be passed as parameters.

4. The scope of the goto statement is limited to its own defining block. Thus, goto's cannot be used for exit from procedures.

5. Standard procedures are limited to I/O only.

6. No empty statement is allowed.

7. The control variable of a for statement must be locally defined.

6.3.2 Extensions

For the sake of efficient execution on byte-oriented machines, an extension to standard type -- 'byte' -- is added. All operations legal for type integer are legal for type byte, too.

Another extension to standard Pascal is automatic type conversion. Some people might think this feature violates certain philosophical aspect of the language Pascal, but I feel obliged to include this feature in Pascal-M for the following reasons.

1. The extension of type byte (short integer) requires automatic conversion between type integer and type byte.

2. There is no function facility in Pascal-M, and hence no standard functions to convert between type real and type integer (e.g. TRUNC, ROUND, etc). Some other way must be provided.

3. Mixed operations between type integer and type real are allowed in standard Pascal anyway.

6.4 DATA TYPES

6.4.1 Type integer

A value of type integer is an element of the implementation-defined subset of whole numbers. In Pascal-M, integers are 2 bytes (16 bits) long, and in 2's complement representation internally. The following arithmetic operators yield an integer value when applied to integer operands.

*	multiply
div	divide and truncate
mod	residue
+	add
-	subtract

Integer operations are guaranteed to be correct only if both operands and result are within [-32768, +32767].

6.4.2 Type byte

A byte, or short integer, is 1 byte (8 bits) long and represented in 2's complement form internally. Any number within the range [-128, +127] could be declared as byte for efficiency. All operations legal for type integer are legal for type byte, and legal for mixed operation between this two types. The result type of mixed operations between type integer and type byte will be of type integer with the following exception: if the dividend is of type byte and divisor is of type integer in div or mod operation, then the result type will be byte.

6.4.3 Type real

A value of type real is an element of the implementation-defined subset of real numbers. In Pascal-M, a real number is 3 bytes long, with 1 byte as characteristic (exponent and sign), and 2 bytes as precision (fraction). The quantity expressed by a real number is the product of the fraction and 2 (not 16 as IBM 360) raised to the power specified by the exponent. The machine form of a real number resembles that of IBM 360 floating point numbers. The leftmost bit of the characteristic is the sign for the real number, the remaining 7 bits are in excess-64 notation, and the 16-bit fraction is an unsigned binary number. Therefore the precision of real number in Pascal-M is about 5 decimal digits, and its range is $(-2^{64}, +2^{64})$, which is about $(-10^{19}, +10^{19})$.

As long as at least one of the operands is of type real (the other operand may be of type byte or type integer), the following operators yield a real value.

- `*` multiply
- `/` divide (even if neither operand is real, the result is always real)
- `+` add
- `-` subtract

Caution: After each real operation, the result (or the intermediate result) will probably be only partially normalized; repetitive operation of some kind might lead to significant loss of precision.

6.4.4 Type Boolean

A Boolean value is one of the logical truth values denoted by the predefined identifiers FALSE and TRUE.

The following logical operators yield a Boolean value when applied to Boolean operands.

- and logical conjunction
- or logical disjunction
- not logical negation

Each of the relational operators (=, <>, <=, <, >=, >) yields a Boolean value. Furthermore, the type Boolean is defined such that FALSE < TRUE. In Pascal-M, a whole byte is used to represent one Boolean value.

6.4.5 Type char

A value of type char is an element of a finite and ordered set of chararacters. In Pascal-M, EBCDIC code is used to represent each character by one byte. Therefore, the collating sequence of chararacters is that of EBCDIC.

6.4.6 Scalar and Subrange Types

Scalar and subrange types are defined as in the Report. In Pascal-M, a value of any scalar or subrange type will be represented by one byte, which implies that the range of subrange types must be no more than 256, and no more than 256 elements are allowed in a scalar type. In section 7.1, the number of elements in scalar types is further restricted.

6.4.7 Array Types

The only data structuring facility included in Pascal-M is array. The elements of arrays in Pascal-M are restricted to having the elementary data types: byte, integer, real, Boolean or char.

Although the number of dimensions of arrays is unlimited, as in standard Pascal, the size of arrays is subject to the constraint of section 3.2.

6.5 STANDARD PROCEDURES: INPUT AND OUTPUT

In Pascal-M, only six input and output procedures are included as standard procedures; only one parameter is passed to each of them. The parameter of each of the three input procedures is called by reference. For the three output procedures, it is called by value. The six I/O procedures are:

READB	Read in a byte; the parameter must of type byte.
READI	Read in an integer; the parameter must of type integer.
READR	Read in a real number; the parameter must of type real.
WRITEB	Print out a byte; the parameter must of type byte, and could be an expression.
WRITEI	Print out an integer; the parameter must of type integer, and could be an expression.
WRITER	Print out a real number; the parameter must of type real, and could be an expression.

6.6 PROGRAMMING EXAMPLES

This section gives some examples of what Pascal-M programs look like. The first example is a simple program that will calculate the GCD (greatest common divisor) of two numbers. This example will be used again in the next chapter to illustrate output format.

```
PROGRAM GCD;
(* THIS PROGRAM WILL FIND THE GCD OF M AND N *)
CONST M=24; N=60;
VAR X,Y: BYTE;
BEGIN
   X:=M; Y:=N;
   WHILE X¬=Y DO
     IF X>Y THEN X:=X-Y
              ELSE Y:=Y-X;
   WRITEB(X)
END.
```

The next example illustrates the use of arrays, a user defined data type, and some other features.

```
(* REF CONWAY & GRIES:'A PRIMER ON PASCAL'  PAGE 268 *)
PROGRAM SORT(INPUT);
  CONST N=10;
  TYPE LIST=ARRAY [1..N] OF INTEGER;
  VAR L:LIST; I,M,T:INTEGER; SORTED:BOOLEAN;
  (* SORT L[1..N] USING BUBBLE SORT *)
  BEGIN
    SORTED:=FALSE;
    M:=N;
    WHILE NOT SORTED AND (M>=2) DO
     BEGIN (* BUBBLE LOOP *)
       (* SWAP L[1..M],PUT LARGEST IN L[M],SET 'SORTED' *)
        SORTED:=TRUE; (* ASSUME L IS SORTED *)
        FOR I:=2 TO M DO
         BEGIN (* SWAP LOOP *)
            IF L[I-1] > L[I] THEN
             BEGIN
                T:=L[I-1];
                L[I-1]:=L[I];
                L[I]:=T;
                SORTED:=FALSE
             END
         END; (* SWAP LOOP *)
        M:=M-1
      END (* BUBBLE LOOP *)
   END.
```

The following program illustrates the use of a user defined data type (in this example, it is scalar type) and the <u>case</u> statement, which are unknown to most other languages.

```
(* THIS PROGRAM COMPUTES THE WEEKLY MILEAGE *)
(* OF MY CAR; I DRIVE TO UNC CAMPUS EVERY   *)
(* MONDAY, WEDNESDAY AND FRIDAY, EACH TRIP  *)
(* (ROUND TRIP) TAKES ME 2 MILES.           *)
(* ON EVERY TUESDAY AND THURSDAY, I DRIVE   *)
(* TO DUKE TO ATTEND CLASS THERE, EACH TRIP *)
(* TAKES ME 28 MILES.                       *)
(* SATURDAY MORNING,I GO TO UNIVERSITY MALL *)
(* FOR SHOPPING, WHICH TAKES ME 10 MILES.   *)
(* SUNDAY NIGHT, I USUALLY VISIT A PAL,     *)
(* THAT TAKES ME 5 MORE MILES TO DRIVE.     *)
PROGRAM:MILECOUNT;
TYPE WEEKDAY=(MONDAY,TUESDAY,WEDNESDAY,
       THURSDAY,FRIDAY,SATURDAY, SUNDAY);
VAR I: WEEKDAY;
    MILEAGE: BYTE;
BEGIN
   MILEAGE:=0;
   FOR I:=MONDAY TO SUNDAY DO
      CASE I OF
         MONDAY,WEDNESDAY,FRIDAY:MILEAGE:=MILEAGE+2;
         TUESDAY,THURSDAY:       MILEAGE:=MILEAGE+28;
         SATURDAY:               MILEAGE:=MILEAGE+10;
         SUNDAY:                 MILEAGE:=MILEAGE+5
      END;(* CASE *)
   WRITER(MILEAGE)
END.
```

The following program illustrates recursive calls and passing of parameters. In this example, N in procedure FACTOR is a call by value parameter; F is a call by reference parameter (the difference is in the presence or absence of VAR on the line of their declaration). There are some subtle points in this example besides the above mentioned ones, e.g. the scope rule of block structured languages, the use of the local variable LOCAL_F to avoid the function facility (which does not exist in Pascal-M), etc.

```
(* THIS PROGRAM COMPUTES THE FACTORIAL OF A NUMBER.*)
PROGRAM MAIN;
  VAR N:BYTE;
      F:INTEGER;
  PROCEDURE FACTOR(N:BYTE; VAR F:INTEGER);
    VAR LOCAL_F:INTEGER;
    BEGIN
      IF N=1 THEN F:=1
      ELSE
       BEGIN
         FACTOR(N-1,LOCAL_F);
         F:=N*LOCAL_F
       END
    END; (* FACTOR *)

  BEGIN (* MAIN PROGRAM *)
    READB(N);
    FACTOR(N,F);
    WRITEI(F)
  END.
```

Chapter 7

THE IMPLEMENTATION

7.1 DIMENSIONAL LIMITS IN PASCAL-M

Dimensional limits are those limits caused by fixed array bounds declared in the compiler itself, and which can be changed easily by simply re-declaring the array bounds in the compiler. The bigger those bounds are, the less constraint the user will feel, but the more main storage will be taken by the compiler during the compilation process.

The dimensional limits in Pascal-M are the following.

1. The maximum number of symbol table entries is 41. This includes 13 predefined symbols (e.g. INTEGER, TRUE, WRITEI, etc.), variable names, procedure names, named and anonymous types defined by the user, labels, constant names. But it does not include the identifiers for defining the components of scalar types.

2. At most 20 different names are allowed in all scalar types in one procedure and all its enclosing procedures (blocks).

3. The maximum length of a single string constant is 20; the maximum length of the sum of all string constants is 25.

4. The number of call by value parameters of a single procedure cannot exceed 8. The number of call by reference parameters of a single procedure cannot exceed 3. The total number of formal parameters of all procedures cannot exceed 10.

5. The sum of all array dimensions that are more than 1 (multidimensional, not vectors) in a procedure and all its enclosing procedures (blocks) cannot exceed 10.

6. An array subscript may be an array element, with maximum nesting 3.

7. The maximum nesting of _if statement_, or _for statement_, or _while statement_, or _repeat statement_, or _case statement_ is 3. The nesting of a certain kind of statement (e.g. _if statement_) does not affect the nesting of any other kind of statement (e.g. _while statement_). For example,

```
IF
  WHILE
    IF                 is permitted,
      WHILE
        IF
          WHILE
```

but

```
IF
  IF
    IF                 is not.
      IF
```

8. The maximum number of case labels in a _case label list_ (not the number of case labels of a case statement, which is unlimited) is 4.

9. The maximum number of _forward goto's_ for a given goto label is 3.

10. No more than 10 recursive calls can appear in the text of all procedures.

7.2 CONSTRAINTS OF TARGET MACHINE

The target machine could be any 8-bit wide microprocessor, but current implementation of Pascal-M will generate code only for SwTPC 6800, a Motorola MCS6800 based microcomputer system designed by Southwest Technical Prods. Corp.

All operations except one in eight-bit microprocessors are performed in eight-bit units. The only exception is that of memory addressing and probably some operations that involve it. With such narrow memory width, any operation involving a data type more than 8 bits wide will have to resort to software solutions. With the limited indexing facility, and no other provision for indirect addresing, this makes the object code very inefficient if no restriction is imposed on the Pascal-M program. Some rules are the inevitable consequences:

1. The stack of activation records will be only 256 bytes long. This implies that no more than 256 bytes are allowed for all variables, and of course that number must be appropriately divided by the maximum number of recursive calls in the program.

2. As a consequence of the above, array size must not exceed 256 bytes. (The size of variables can be calculated from elementary data type sizes as described in section 6.4.)

7.3 JOB CONTROL FOR RUNNING A PASCAL-M PROGRAM

7.3.1 Structure of a Run

A Pascal-M program is cross-compiled at TUCC. The minimum JCL required to compile a Pascal-M program at TUCC is as follows:

```
// (JOB CARD)
//*PASSWORD
//  EXEC PASCALM
//C.SYSIN DD (data set with your source program)
//G.HEXCODE DD (data set that will hold the machine code)
```

The above JCL will give you:

1. A source list of your Pascal program with line numbers added.

2. Error messages if any.

3. A combined attributes and cross reference table.

4. Target machine code in hex, if your program has no error.

If you specify a dataset instead of the printer (SYSOUT=A) for the file G.HEXCODE, you will get a machine readable form of object code.

If you want a trace of shifts and reductions of the compilation process, specify after the //C.SYSIN card:

```
//C.TRACE DD SYSOUT=A
```

If you do not want the attributes and cross reference table, then specify after SYSIN card, or if you have TRACE card, specify after the TRACE card:

```
//C.XREF DD DUMMY
```

If you want a printout of the intermediate code generated by compiling your program, specify after SYSIN card, and after TRACE card and XREF card if any, the following:

```
//C.ICODE DD SYSOUT=A
```

7.3.2 Program Format

1. The standard field for source statements is columns 2 through 80.

2. The standard position for carriage control for the listing of the source program is position 1. Only five of the USASI codes are recognized for this purpose:

blank	space 1 line before printing (normal printing)
0	space 2 lines before printing
1	skip to channel 1 (page eject)
-	space 3 lines before printing
+	do not space before printing (overprinting)

   Carriage control characters do not appear on the source listing. If any character other than these five appears in position 1, Pascal-M assumes that the user neglected to skip position 1 and the scan will begin in position 1. A warning message will be issued.

7.3.3 Using an object module

The code generated by the Pascal-M compiler is an object module that can be run under the SWTBUG operating system [21].

To load an object program of Pascal-M, first the user must load the library routines from a floppy disk labelled 'Pascal-M' into the RAM of the SwTPC 6800:

Step 1: Hook up the HP2645 terminal to the SwTPC, turn the duplex switch to the 'full' position.

Step 2: Power on the HP terminal, the microcomputer, and the floppy disk system (The floppy disk system is issued by Smoke Signal Broadcasting [20]).

Step 3: Put the floppy disk labelled 'DOS-68' into disk drive unit 0; put the floppy disk labelled 'Pascal-M' into unit 1.

Step 4: Wait until SWTBUG prompts you with a dollar sign, then enter the command: J 8020.

Step 5: Wait until the message: DOS-68 appears; enter the command: GET,1:PASCAL
After this step the library routines will be loaded into RAM.

Step 6: Wait until the DOS prompts you with greater than sign, then enter command: GET,1:LOADER.
after this step, the absolute loader will be in.

Step 7: Turn the duplex switch to the 'half' position. Sign on to TSO (don't power off the SwTPC 6800), QED and list the dataset that contains the hex code. Leave the code on the HP screen.

Step 8: Logoff from TSO and switch back to the SwTPC 6800.

Step 9: Hit reset of the microcomputer and get a dollar sign from SWTBUG, then enter the command:
J 0000
This step transfers control to the loader.

Step 10: Turn the HP terminal to 'block mode', and enter each line of code on the screen by hitting the 'enter' key.

Step 11: After all lines have been entered, hit the reset again.

Step 12: After a dollar sign appears, the program should be in RAM. Now enter the command: J 0100 and your program will start running.

7.3.4 Input and Output

Input and output are performed interactively on the primary I/O device of the microcomputer system by calling the standard procedures of section 6.5. The primary I/O device of the Southwest Tech microcomputer system is the console terminal. In particular, the microcomputer installed in Phillips 273 uses an HP2645 as the console terminal.

7.4 COMPILATION OUTPUT FORMAT

7.4.1 Source listing

After compiling the Pascal-M program, the compiler will list the Pascal-M program with line numbers added. The following is the source listing of the GCD program of section 6.6.

```
  1     PROGRAM GCD;
  2     (* THIS PROGRAM WILL FIND THE GCD OF M AND N *)
  3     CONST M=24; N=60;
  4     VAR X,Y: BYTE;
  5     BEGIN
  6        X:=M; Y:=N;
  7        WHILE X¬=Y DO
  8          IF X>Y THEN X:=X-Y
  9                 ELSE Y:=Y-X;
 10        WRITEB(X)
 11     END.
```

7.4.2 Cross reference and attribute table

The combined cross reference and attribute table is actually a dump of the symbol table with cross references of symbols added.

The cross reference portion lists each identifier, the number of the line and name of the procedure where the identifier is defined, and the line number associated with each occurrence of the identifier. The attribute portion lists the value of each field in the symbol table entry.

The following is an example of the cross reference and attribute table for the GCD program of section 6.6. The first three columns list the number of line on which a symbol is defined (ref. section 7.4.1), the symbol itself, and the name of the procedure where the symbol is defined, respectively. The next column, labelled SYMTYPE, describes the type of the symbol; the codes are:

- 91 constant identifier
- 92 function identifier (not implemented)
- 93 variable identifier
- 94 field identifier (not implemented)
- 95 type identifier

The next column, VALUE1, roughly corresponds to the values of symbols; a minus one in that field usually denotes that value is undefined at compilation time. The last column, OFFSIZE, gives the offset within the activation record. The offset of each symbol is indispensible in understanding the intermediate code (see section 3.4.4). The second line of each symbol entry lists the line number associated with each occurrence of that symbol.

	SYMID	PROC ID	SYMTYPE	VALUE1	OFFSIZE
3	M	GCD	91	24	0
		6,			
3	N	GCD	91	60	1
		6,			
4	X	GCD	93	-1	2
		6, 7, 8, 8, 8, 9, 10,			
4	Y	GCD	93	-1	3
		6, 7, 8, 8, 9, 9,			

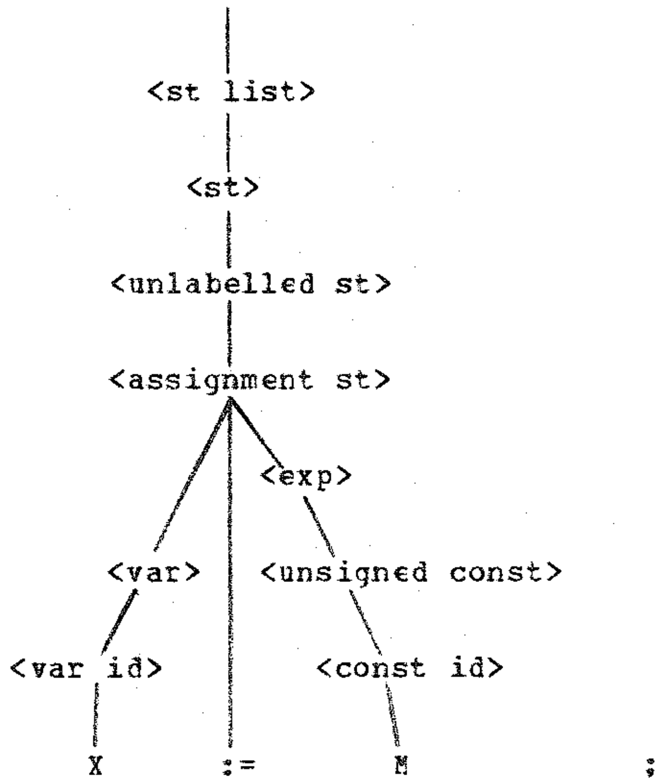
The above explanation is incomplete; besides, some other fields in the symbol table will be printed out too; but their major use is for the maintenance of the compiler rather than for aiding the user to debug. The interested user is referred to Pascal-M Program Logic Manual (Appendix B).

7.4.3 Trace of compilation

Because LR(1) parsing was used for processing the statement part of a program in this compiler (recursive descent was used for parsing the declaration part of a program), it is very easy to construct the parse tree for statements by tracing the parsing actions (shift and reduce). This can be done by specifying the TRACE DD card as described in section 7.3. The following is part of the trace produced during compilation of program GCD of the previous chapter. On the next page, a partial tree is constructed according to the first 11 lines of the following trace and shows that it is a useful debugging aid.

```
SHIFT TO STATE   4, NEW SYM= 93
REDUCE  18              VAR--VAR ID, VTYPE=0
SHIFT TO STATE   8, NEW SYM=121
SHIFT TO STATE  22, NEW SYM= 91
REDUCE  42              UNSIGNED CONST--CONST ID
REDUCE  38              EXP--UNSIGNED CONST
SHIFT TO STATE  43, NEW SYM=109
L_RD    16              ASSGN ST--VAR := EXP
REDUCE   5              UNLABELLED ST--ASSGN ST
REDUCE   2              ST -- UNLABELLED ST
REDUCE  61              ST LIST--ST
```

```
SHIFT TO STATE   5, NEW SYM=109
REDUCE  62              <;>--;
SHIFT TO STATE  20, NEW SYM= 93
REDUCE  18              VAR--VAR ID, VTYPE=0
SHIFT TO STATE   8, NEW SYM=121
SHIFT TO STATE  22, NEW SYM= 91
REDUCE  42              UNSIGNED CONST--CONST ID
REDUCE  38              EXP--UNSIGNED CONST
SHIFT TO STATE  43, NEW SYM=109
L_RD    16              ASSGN ST--VAR := EXP
REDUCE   5              UNLABELLED ST--ASSGN ST
REDUCE   2              ST -- UNLABELLED ST
REDUCE  60              ST LIST--ST LIST <;> ST
SHIFT TO STATE   5, NEW SYM=109
REDUCE  62              <;>--;
SHIFT TO STATE  20, NEW SYM= 34
REDUCE  83              <WHILE>--WHILE
SHIFT TO STATE  15, NEW SYM= 93
REDUCE  18              VAR--VAR ID, VTYPE=0
SHIFT TO STATE  30, NEW SYM=111
L_RD    34              <EXP>-- <VAR>
SHIFT TO STATE  37, NEW SYM=111
```



7.4.4 Intermediate code

Intermediate code of Pascal-M programs can be obtained by specifying the ICODE DD card as described in section 7.3. The following example is the intermediate code of program GCD of section 2.6. The last column is commentary appended to the intermediate code. For the definition of the intermediate code, the user is referred to Pascal-M, Program Logic Manual (Appendix B).

```
  1 MARK  0   4     mark activation record with 4 bytes
  2 X     0   3     do nothing
  3 LITB  0  24     load literal 24
  4 STB   0   2     store 24 to X (refer to the cross
                    reference table of section 7.4.2)
  5 LITB  0  60     load literal 60
  6 STB   0   3     store 60 to Y
  7 LODB  0   2     load X
  8 LODB  0   3     load Y
  9 EQUB  0   0     if X=Y ?
 10 X     0   0     do nothing
 11 JPF   0  26     jump false to 26
 12 LODB  0   2     load X
 13 LODB  0   3     load Y
 14 GTB   0   0     if X>Y ?
 15 JPF   0  21     jump false to 21
 16 LODB  0   2     load X
 17 LODB  0   3     load Y
 18 SUBB  0   0     X-Y
```

```
19 STB    0   2      X:=X-Y
20 JUMP   0  25
21 LODB   0   3      load Y
22 LODB   0   2      load X
23 SUBB   0   0      Y-X
24 STB    0   3      Y:=Y-X
25 JUMP   0   7
26 LODB   0   2      load X
27 MARK   0   3      mark activation record with 3 bytes
28 STB    0   2      store X to print buffer
29 CALL   0 -10      call WRITEB, write out X
30 RTS    0   0      return
```

7.4.5 Error messages

All syntactic errors, all attempts to use unimplemented features, and all semantic errors that it is possible to detect at compilation time will be reported. No run-time checking is provided, however.

Since a description of the error is printed instead of an error code for each error encountered, a list of all error messages is not included in this manual.

Examples of error messages:

LINE	ERROR MESSAGE
3	'[' EXPECTED
4	'END' EXPECTED
11	2ND OPERAND OF 'AND' OPERATION SHOULD BE OF TYPE BOOLEAN.

7.4.6 Machine codes

The object module (6800 machine code) will be in the dataset specified by the DD card G.HEXCODE. Each line of code is in absolute format, and is composed of 40 bytes, each byte is in 2 hex digits and therefore the 40 bytes will occupy the whole 80 columns, except possibly for the last line.

Appendix B

PROGRAM LOGIC MANUAL

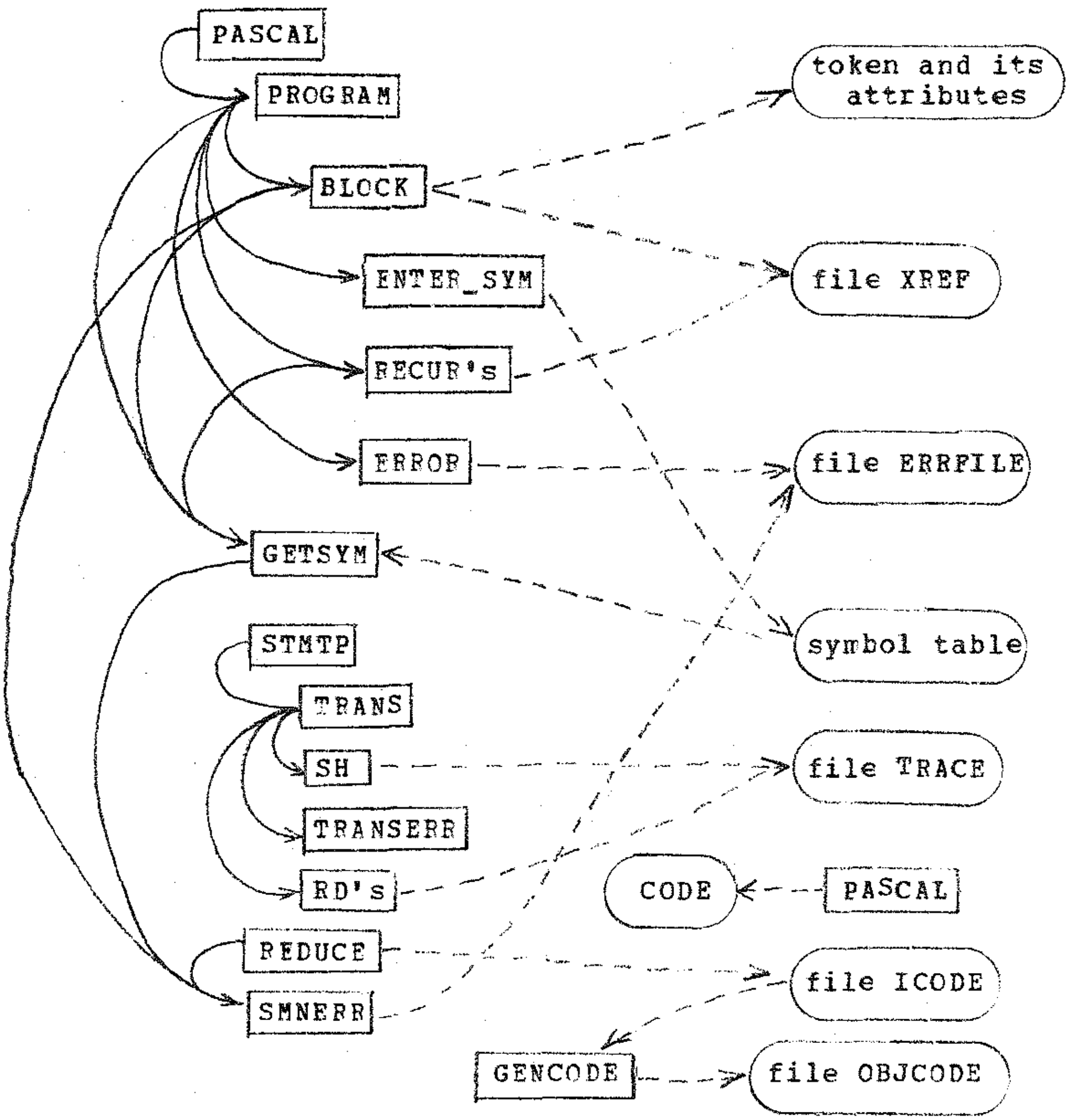
Chapter 8

PROCEDURE STRUCTURE

The whole compiler is divided into 6 external procedures, each of which might have some internal procedures. The static nesting of all the procedures is diagrammed on this page. The dynamic structure of the procedures and the interconnections amoung them and some important data and files are shown on the next page. The function of each procedure is discussed on the following pages.

```
PASCAL
   PROGRAM
      BLOCK
         ENTER_SYM
         TYPE
            SIMPLE_TYPE
         CONST
         FIELD_LIST
         PARM_LIST
         ERROR

GETSYM
   GETCH

STMTP
   TRANS
   SH
   RD
   LRD
   TRANSERR

REDUCE
   GEN
   PATCH

SMNERR

GENCODE
   GENM
   HEX
   LOADA
   LEVOFF
```

Static structure of procedures



Dynamic structure of procedures and data

The solid arrows in the above graph mean procedure call, the dotted arrows mean 'production' or 'use' of data depending on the direction of the arrow. The boxes represent procedures, the circles represent data or files. The RECUR's on the above graph abbreviates the set of recursive procedures inside procedure BLOCK. RD's abbreviates for procedures RD and LRD. Some procedures shown on the static structure but not on the dynamic one are small procedures which are called only by their own mother procedure and have no outside connection.

The procedure PASCAL is the main program for the first five external procedures; it initializes the symbol table and most variables and calls procedure PROGRAM to start parsing. After the return from PROGRAM, the source program should all be processed and intermediate code should be ready to be written onto file PCODE for further processing by procdeure GENCODE. Should any error occur during parsing, a return code RETCD will be set to 99, which will inhibit invoking GENCODE.

The procedure PROGRAM is the top (or outer-most) procedure of a series of recursive procedures. A Pascal program is composed of a <program heading> and a <block>; a <block> is divided into 2 parts, <declaration part> and <statement part>, where within a <declaration part> there might be some more <block>'s and the cycle goes on. In this compiler recursive descent parsing is used to parse the <declaration part>, procedure PROGRAM corresponds to the nonterminal <program> in the grammar and it calls procedure BLOCK, which corresponds to nonterminal <block>. The procedure BLOCK maintains the symbol table and calls internal procedures TYPE, CONST, FIELD_LIST, and PARM_LIST, each of which corresponds to a nonterminal in the grammar and all of which help BLOCK maintain the symbol table. Internal procedure ENTER_SYM is called whenever a symbol and its attributes are to be entered into the symbol table. Whenever an error is encountered during parsing the declaration part, internal procedure ERROR is called and a set of possible follow up tokens (in array FSYM) are passed to it so that some types of error recovery are possible. Besides trying to recover from the error, procedure ERROR reports the error by writing a message onto the file ERRFILE and setting a flag ERRFLAG, which will later inhibit invoking object code generation routine GENCODE.

The procedure GETSYM is the lexical scanner, which performs the following tasks.

1. Reads in a token and determines its meaning. The meaning of the token is returned through an external variable SYM. (The encoding of tokens is listed in following sections.)

2. Returns the value of the token, if it is a literal, through external variables VAL1, VAL2, or STRING.

3. Returns the name of the token, if the token is an identifier, through external variable ID.

4. Prints out the source program as it is read in, and maintains the cross reference table.

It has an internal procedure GETCH, which is called to read next character from the source program into variable CH.

The procedure STMTP is the main routine of <statement part>. It is called once for each <block> of the source program by procedure BLOCK, and then it takes over the parsing of <statement part> until its end. The parsing algorithm used in this part is SLR(1), and a parser generator was used to build the decision table for parsing actions. Details will be discussed in section 10.4. It has 5 internal procedures: TRANS, SH, RD, LRD, and TRANSERR. Procedure TRANS is actually the decision table for parsing actions for <statement part>. It examines the top element of a state stack STSTK and current token SYM and performs one of the following 4 actions.

1. Calls procedure SH, which shifts to a new state by stacking a new state on the state stack and gets a new token.

2. Calls procedure RD, which calls procedure REDUCE to perform the necessary semantic action.

3. Calls procedure LRD, which is almost the same as RD, the only difference being that procedure TRANS looked ahead one token before it calls LRD.

4. Calls procedure TRANSERR whenever an illegal combination of top of STSTK and current token SYM is encountered (i.e. no table entry for this combination). Procedure TRANSERR reports at which state a transition error is encountered and calls SH to go on.

The procedure REDUCE is a collection of semantic routines of <statement part>. It is called whenever a reduction action is to be performed during parsing <statement part>. The production number of which reduction is to be performed is passed to this procedure through variable PROD; the procedure selects the corresponding semantic routine and generates intermediate code accordingly. This procedure also pops out some tokens from the syntax stack STSTK of procedure STMTP by decreasing the variable TOP, which is the pointer to the top of the syntax stack. Two internal procedures GEN and PATCH are used by REDUCE. Procedure GEN generates and maintains the array of intermediate code CODE, which is a structured array variable with 3 fields: OPCODE, CODELEV, and CODEOFF; procedure PATCH patches forward references whenever these destinations become known.

The procedure SMNERR is the routine to report some common syntax errors encountered during parsing <statement part> (e.g. confusing '[' with '('), and all semantic errors during any part of compilation (e.g. wrong operand type for a certain operation). It does not attempt to correct the error; it simply sets the flag ERRFIAG as procedure ERROR does, and continues parsing.

The procedure GENCODE is a phase separate from the foregoing five external procedures because it is source-program-independent. It takes the intermediate code generated by procedure PASCAL and optimizes it somewhat, then generates object code from the optimized intermediate code. Details of procedure GENCODE will be discussed in chapter 12.

All the above procedures are written in PL/I, and were compiled by OS PL/I Checkout Compiler, version 1, release 3.0, PTF 26. (The PL/I Optimizing Compiler, version 1, release 3.0, PTF 65 has some bugs in translation of certain SELECT statements. These bugs kept me from using the Optimizing Compiler.)

Chapter 9

LEXICAL ANALYSIS

Lexical analysis is done by procedure GETSYM as described in chapter 1: a token is read in and analyzed and encoded into a number, then it is returned to the calling procedure through the external variable SYM. The encodings of tokens used in lexical analysis are listed in the following sections.

9.1 RESERVED WORDS

The 35 reserved words are encoded as 1, 2, 3, ..., 35, respectively, according to their alphabetic order.

rsw	SYM	rsw	SYM	rsw	SYM
AND	1	FUNCTION	13	PROGRAM	25
ARRAY	2	GOTO	14	RECORD	26
BEGIN	3	IF	15	REPEAT	27
CASE	4	IN	16	SET	28
CONST	5	LABEL	17	THEN	29
DIV	6	MOD	18	TO	30
DO	7	NIL	19	TYPE	31
DOWNTO	8	NOT	20	UNTIL	32
ELSE	9	OF	21	VAR	33
END	10	OR	22	WHILE	34
FILE	11	PACKED	23	WITH	35
FOR	12	PROCEDUR	24		

9.2 OPERATORS

token	SYM	token	SYM
+	115	-	114
*	113	/	112
=	111	,	110
;	109	@	108
(	104	)	105
[	107	]	106
>	118	<	122
<=	123	>=	119
<>	124	¬=	124

:	120	.	126
:=	121	..	127

9.3 OTHER ENCODINGS

unsigned integer	100
unsigned real	101
string	103
type identifier	90
const identifier	91
function identifier	92
variable identifier	93
field identifier	94
procedure identifier	95

Chapter 10

SYNTAX ANALYSIS

Because the whole compiler is a *syntax driven* translation process, syntax analysis, besides its apparent major function: *syntax checking* (or loosely, *parsing*), has 2 other important functions:

1. building the symbol table; and
2. driving semantic routines.

The main component of a PASCAL program is a <block>, which is subdivided into 2 parts corresponds to the division of labor of processing the 2 important functions of syntax analysis:

1. <declaration part>, parsed by recursive descent; and
2. <statement part>, parsed by SLR(1).

10.1 ENCODING OF NONTERMINALS

Recursive descent is a top-down parsing method, in which each nonterminal symbol corresponds to a procedure instead of being merely a token. Since the <declaration part> is parsed by recursive descent, all the nonterminals in this part are not encoded into numbers. However, for a decision-table-driven bottom-up parsing method like the family of LR parsing methods, each nonterminal does not act too differently from a terminal token. In order to enable the decision table to have uniform input, the nonterminals must be encoded into simple numbers as terminal tokens are. The following page shows the list of nonterminals of <statement part> and their codes.

nonterminal	SYM	nonterminal	SYM
<SYSTEM GS>	201	<ST P>	202
<COMPOUND ST>	203	<ST>	204
<UNLABELLED ST>	205	<LABEL>	206
<ASSGN ST>	207	<PROC ST>	208
<GOTO ST>	209	<EMPTY ST>	210
<IF ST>	211	<CASE ST>	212
<REPEAT ST>	213	<WHILE ST>	214
<FOR ST>	215	<WITH ST>	216
<VAR>	217	<EXP>	218
<INDEXED VAR>	219	<ELIST>	220
<ARRAY VAR>	221	<FUN DESIGNATOR>	222
<SET>	223	<UNSIGNED CONST>	224
<ACTL PARM LIST>	225	<ACTL PARM>	226
<ELEMENT LIST>	227	<ELEMENT>	228
<ST LIST>	229	<;>	230
<THEN>	231	<ELSE>	232
<OF>	233	<CASE LIST E LIST>	234
<CASE LIST ELEMENT>	235	<CASE LABEL LIST>	236
<:>	237	<CASE LABEL>	238
<CONST>	239	<WHILE>	240
<DO>	241	<REPEAT>	242
<UNTIL>	243	<CONTROL VAR>	244
<INITIAL VALUE>	245	<FINAL VALUE>	246
<RECORD VAR LIST>	247		

10.2 SYMBOL TABLE

The data structure of the symbol table is declared as follows.

```
1 SYMTBL(1:40) EXT,
  2 SYMID CHAR(8),
  2 SYMLEV FIXED BIN,
  2 SYMTYPE FIXED BIN,
  2 SUBTYPE FIXED BIN,
  2 VALUE1  FIXED BIN,
  2 VALUE2  FIXED BIN,
  2 OFFSIZE FIXED BIN;
```

The interpretation of each field in SYMTBL is dependent upon the value of SYMTYPE:

Case SYMTYPE of

90 type id
SYMID is the name of this identifier.
SYMLEV is the static block level on which this identifier is declared.
case SUBTYPE of
0: scalar byte, OFFSIZE is 1(byte).

1: scalar integer, OFFSIZE is 2(bytes).
2: scalar real, OFFSIZE is 3(bytes).
3: scalar boolean, OFFSIZE is 1(byte).
4: scalar character, OFFSIZE is 1(byte).
5: subrange type, the range (implemented as one byte) is (VALUE1..VALUE2), OFFSIZE is 1(byte).
6,7,8,9,10: arrays: byte, integer, real, boolean, character respectively. VALUE1 is the pointer to dope vector (DOPE), for 1-dimensional arrays, it is:
(base location of vector) - (lower bound) * (element size)
VALUE2 is the number of dimensions.

91: constant id
SYMID and SYMLEV have the same interpretation as for type id.
case SUBTYPE of
0: byte, value is in 2nd byte of VALUE1.
1: integer, value is in VALUE1.
2: real, fraction is in VALUE1, characteristic is in the 2nd byte of VALUE2.
3: TRUE(VALUE1=1) or FALSE(VALUE1=0).
4: character, value is in 2nd byte of VALUE1.
11:string, VALUE1 is the pointer to STRAREA, VALUE2 is the length of string.

92: function id (not implemented)

93: variable id
SYMID and SYMLEV have the same interpretation as for type id.
case SUBTYPE of
0: scalar byte
1: scalar integer
2: scalar real
3: scalar boolean
4: scalar character
other: index of type id in symbol table.
If this variable is a formal parameter called by reference then VALUE1=0; otherwise VALUE1=-1.
VALUE2 is not used.

94: field id (not implemented)

95: procedure id
SYMID is the procedure name.
SYMLEV is the address in P-code of this procedure; if negative then it is a builtin procedure.
VALUE1 is the pointer to parameter information (PARAMNFO).
VALUE2 is the number of formal parameters of this procedure.
OFFSIZE is the size of the activation record (in bytes).

96: label
SYMID is unused.
SYMLEV is the numeric value of the label.
case subtype of
0: undefined; VALUE1, VALUE2 and OFFSIZE are the list of undefined goto's (therefore at most 3 forward references are allowed).
1: defined; OFFSIZE is the P-code address of label definition.

The symbol table is initialized by the following 13 predefined words.

Index	SYMID	SYMTYPE	SUBTYPE	OFFSIZE	VALUE1	VALUE2
0	BYTE	90	0	1	-1	-1
1	INTEGER	90	1	2	-1	-1
2	REAL	90	2	3	-1	-1
3	BOOLEAN	90	3	1	-1	-1
4	CHAR	90	4	1	-1	-1
5	TRUE	91	3	1	1	-1
6	FALSE	91	3	1	0	-1
7	READB	95	-7	4	1	1
8	READI	95	-8	4	1	1
9	READR	95	-9	4	1	1
10	WRITEB	95	-10	3	2	1
11	WRITEI	95	-11	4	3	1
12	WRITEI	95	-12	5	4	1

Besides SYMTBL, there are some auxiliary data structures to the symbol table, declared as follows.

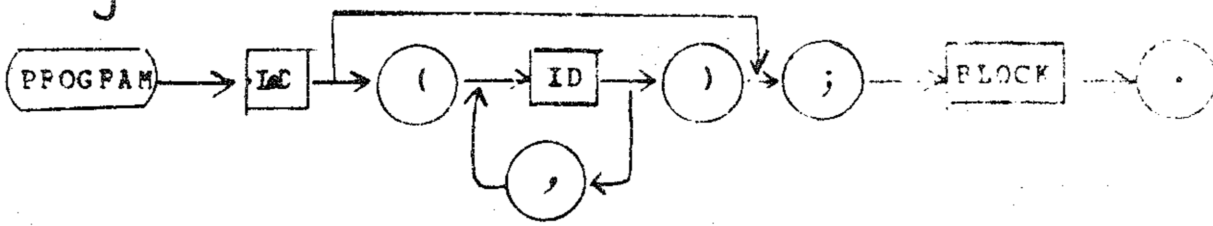
```
LNDEFD(13:40) FIXED BIN, /* On which lines the symbol */
                         /* is defined.               */
LNSAPPR(13:40) CHAR(80)  /* On which lines the symbol */
               VAR EXT,  /* appears (only the 1st 20  */
                         /* appearances are included).*/
DOPE(1:10) FIXED BIN EXT,/*Dope vector, for each array*/
                         /*information in it is:      */
                         /* CONSTP, D2, D3, ... Dn    */
                         /* where Di is the i'th dim'n*/
DPX FIXED BIN INIT(0),   /* Index of DOPE.            */
STRAREA CHAR(25) VAR EXT,/* String area.              */
PMI FIXED BIN,           /* Index of PARMNFO.         */
1 PARMNFO(1:10) EXT,     /* Parameter information.    */
  2 POFF FIXED BIN,      /* Offset of each formal parm*/
  2 PREF BIT(1),         /* Is it a call by ref parm? */
  2 PTYPE FIXED BIN,     /* Type of the parameter     */
TX FIXED BIN EXT,        /* Index of SYMTBL of last   */
                         /* entered identifier.       */
SYMX FIXED BIN EXT,      /* Index of SYMTBL of last   */
                         /* encountered identifier.   */
```

10.3 DECLARATION PART

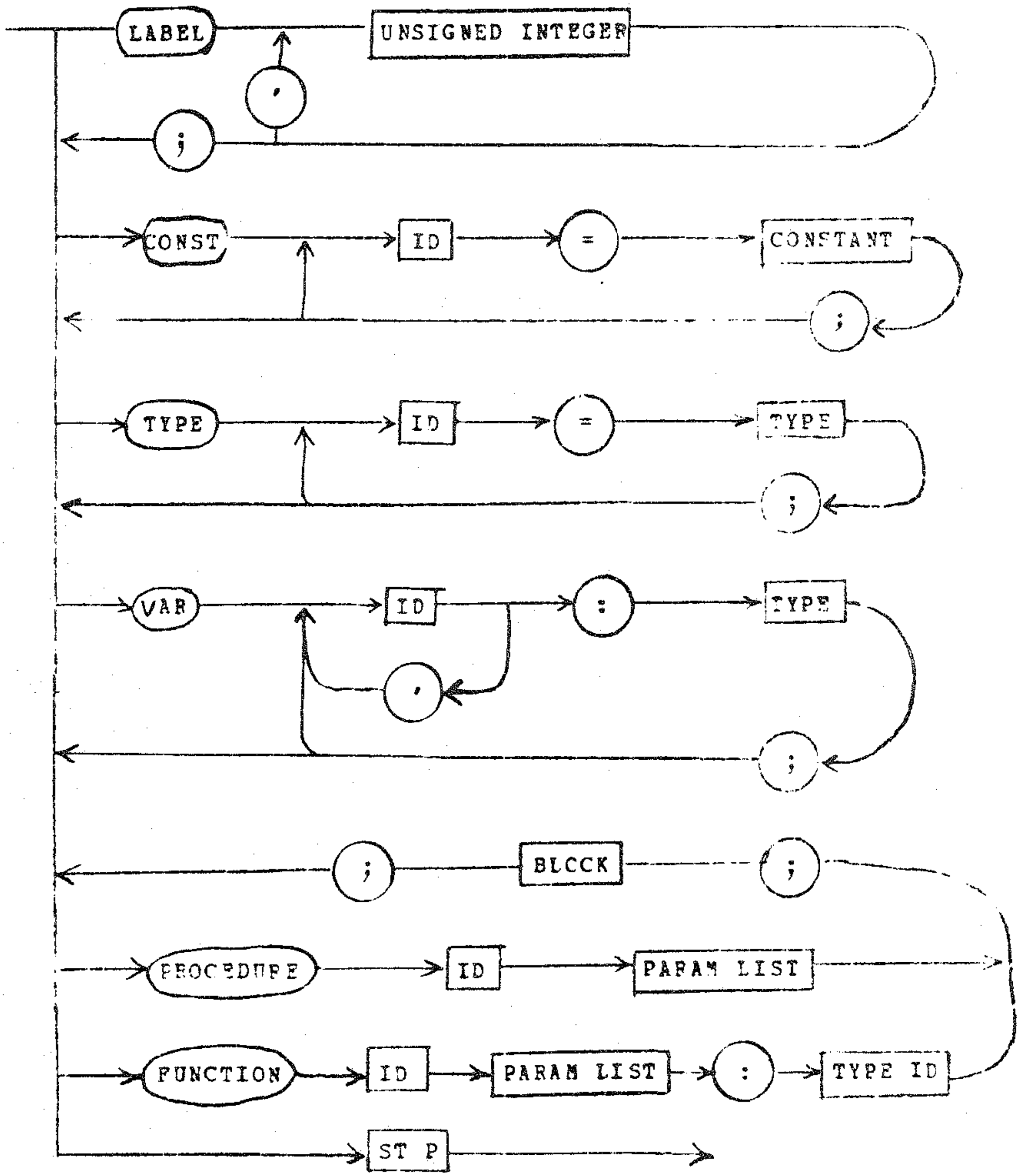
10.3.1 Grammar

Because recursive descent is used in parsing this part, it is more convenient to express this part of the grammar by syntax diagrams than by BNF.

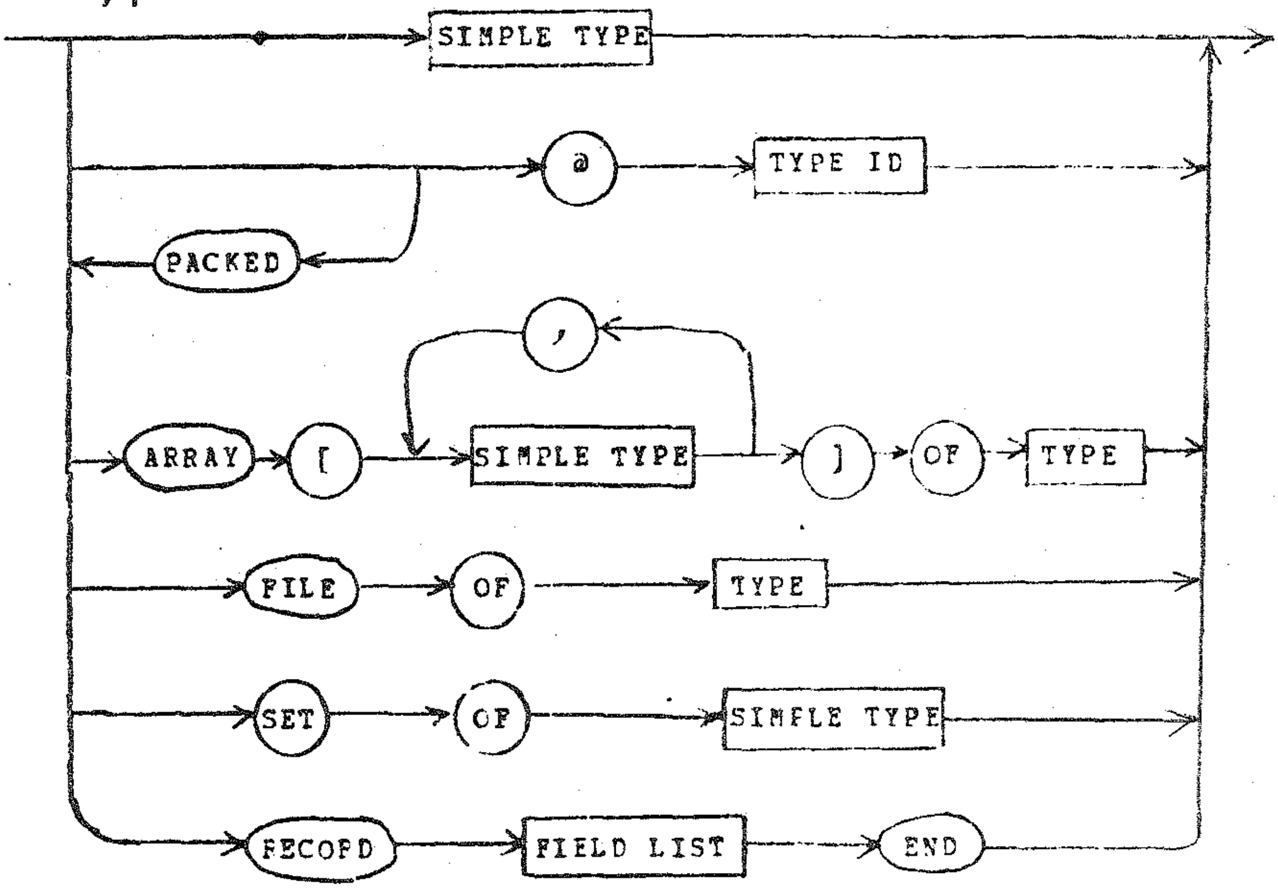
Program



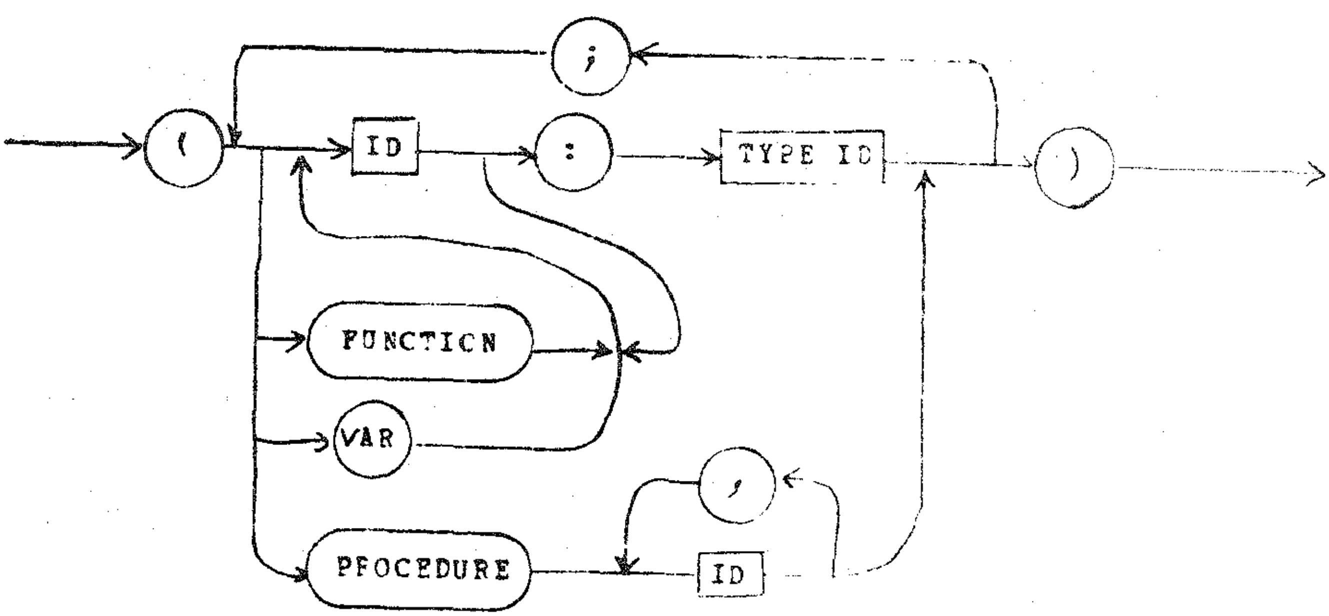
Block



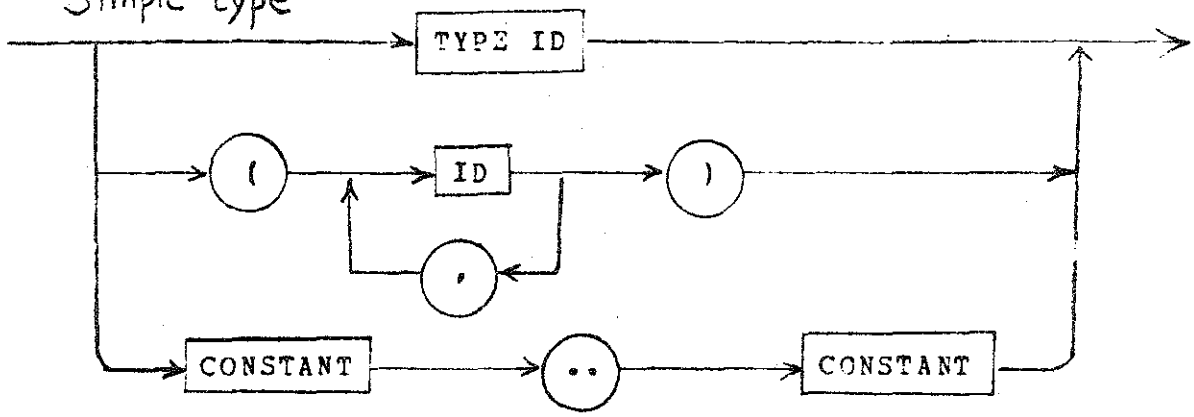
Type



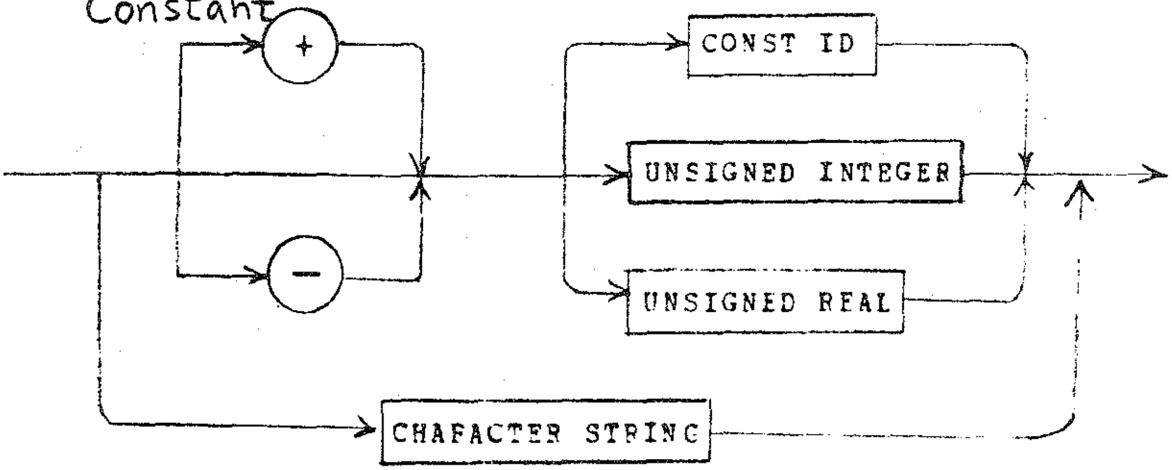
PARAM LIST



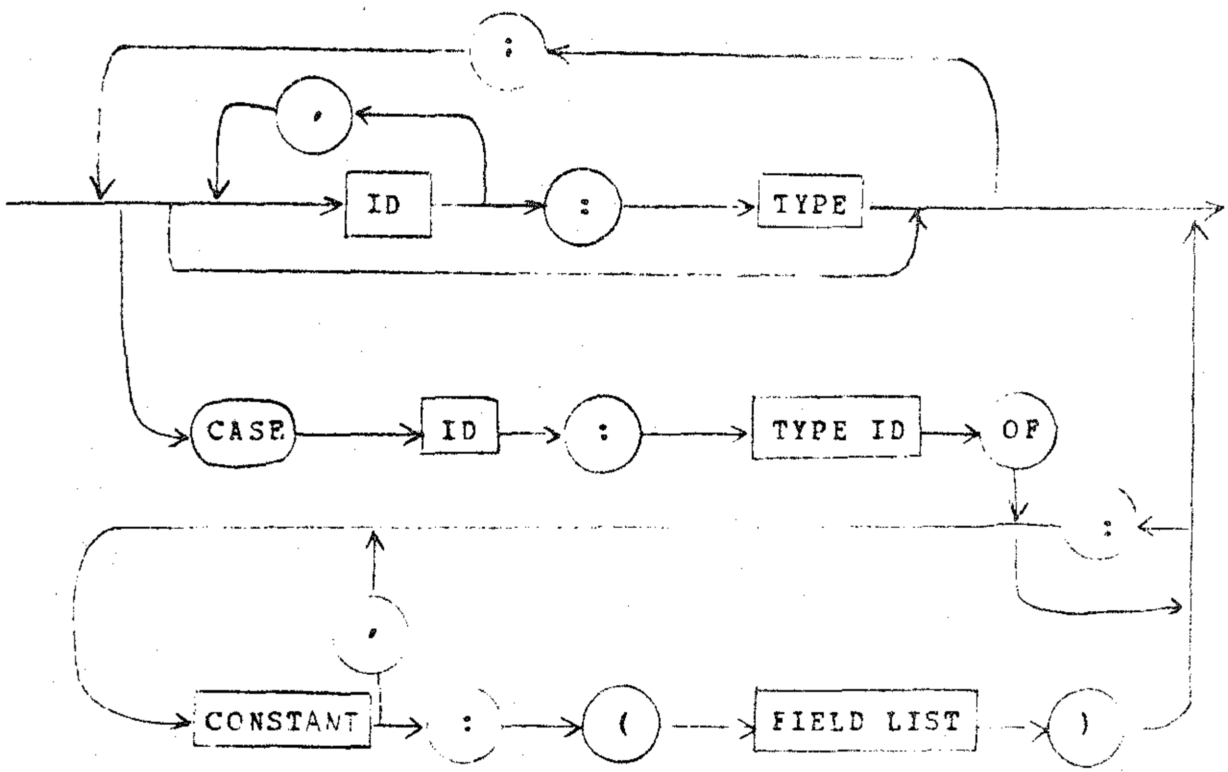
Simple type



Constant



FIELD LIST



10.3.2 Error recovery

The procedure ERROR will report syntax errors encountered during parsing the declaration part, and it will recover from 3 common types of error.

If the correct sequence of symbols is

. . A B C . .

and the parser is expecting to process symbol 'B' while an error is encountered, the procedure ERROR recovers from the following 3 types of error:

1. missing current symbol, while the next is right,
   e.g. . . A C . .

2. wrong symbol is encountered, but the next symbol is right,
   e.g. . . A D C . .

3. an illegal symbol is inserted before the correct symbol,
   e.g. . . A D B X . .

The algorithm used here is simple. It is centered around the array FSYM, follow-up symbols, which is the set of possible next symbols. If symbol A has been read and the parser is expecting to read symbol B but reads something else as the current symbol, then the following actions will be taken.

step 1. report error

step 2. check if the current symbol is in the follow-up symbols. If so, then assume a symbol is missing, return; otherwise continue to step 3.

step 3. read in the next symbol and see if it is the symbol which had been expected as the current symbol. If true, then assume an illegal symbol is inserted before the correct one, return; otherwise continue to step 4.

step 4. check to see if the new symbol is in the follow-up symbols. If so, then assume the current symbol is wrong (error type 1), flag DCGSYM to scanner. (Because next SYM is already being read in, the next call to GETSYM will be a return only.)

10.4 STATEMENT PART

SLR(1) was used in parsing <statement part>. A parser generator (written by R. Wetfer at Cornell University) was used to produce the decision table for parsing actions. The input to the parser generator (the BNF grammar for <statement part>) is listed on the following 2 pages. The output of the parser generator (the decision table) is coded into procedure TRANS. The actions have been briefly discussed in chapter 8; the principle of this parsing technique can be found in [1], [2], [10].

Whenever a reduction action is called, be it RD or LRD, one of the semantic routines in procedure REDUCE will come into play and some pieces of intermediate code are generated. The set of intermediate code used in the compiler is defined in chapter 11.

```
<ST P> <COMPOUND ST>
<ST> <UNLABELLED ST>
    <LABEL> : <UNLABELLED ST>
<LABEL> <UNSIGNED INTEGER>
<UNLABELLED ST> <ASSGN ST>
                <PROC ST>
                <GOTO ST>
                <EMPTY ST>
                <COMPOUND ST>
                <IF ST>
                <CASE ST>
                <REPEAT ST>
                <WHILE ST>
                <FOR ST>
                <WITH ST>
<ASSGN ST> <VAR> := <EXP>
           <FUN ID> := <EXP>
<VAR> <VAR ID>
      <INDEXED VAR>
      <VAR> . <FIELD ID>
      <VAR> @
<INDEXED VAR> <ELIST> ]
<ELIST> <ELIST> , <EXP>
        <ARRAY VAR> [ <EXP>
<ARRAY VAR> <VAR>
<EXP> <EXP> <RELATIONAL OP> <EXP>
      <EXP> + <EXP>
      <EXP> - <EXP>
      <EXP> OR <EXP>
      + <EXP>
      - <EXP>
      <EXP> <MULTIPLYING OP> <EXP>
      ( <EXP> )
      <VAR>
      <FUN DESIGNATOR>
      <SET>
      NOT <EXP>
      <UNSIGNED CONST>
<UNSIGNED CONST> <UNSIGNED INTEGER>
                 <UNSIGNED REAL>
                 <STRING>
                 <CONST ID>
                 NIL
<FUN DESIGNATOR> <FUN ID>
                 <FUN ID> ( <ACTUAL PARAM LIST> )
<ACTUAL PARAM LIST> <ACTUAL PARAM LIST> , <ACTUAL PARAM>
                    <ACTUAL PARAM>
<SET> [ <ELEMENT LIST> ]
      [  ]
<ELEMENT LIST> <ELEMENT>
               <ELEMENT LIST> , <ELEMENT>
<ELEMENT> <EXP>
          <EXP> .. <EXP>
```

```
<PROC ST> <PROC ID>
          <PROC ID> ( <ACTUAL PARAM LIST> )
<ACTUAL PARAM> <EXP>
               <PROC ID>
<GOTO ST> GOTO <UNSIGNED INTEGER>
<EMPTY ST>
<COMPOUND ST> BEGIN <ST LIST> END
<ST LIST> <ST LIST> <;> <ST>
          <ST>
<;> ;
<IF ST> IF <EXP> <THEN> <ST>
        IF <EXP> <THEN> <ST> <ELSE> <ST>
<THEN> THEN
<ELSE> ELSE
<CASE ST> CASE <EXP> <OF> <CASE LIST E LIST> END
<OF> OF
<CASE LIST E LIST> <CASE LIST E LIST> ;<CASE LIST ELEMENT>
                   <CASE LIST ELEMENT>
<CASE LIST ELEMENT> <CASE LABEL LIST> <:> <ST>
                    <EMPTY ST>
<:> :
<CASE LABEL LIST> <CASE LABEL LIST> , <CASE LABEL>
                  <CASE LABEL>
<CASE LABEL> <CONST>
<WHILE ST> <WHILE> <EXP> <DO> <ST>
<REPEAT ST> <REPEAT> <ST LIST> <UNTIL> <EXP>
<FOR ST> FOR <CONTROL VAR> := <INITIAL VALUE> TO
(continue)                   <FINAL VALUE> <DO> <ST>
<FOR ST> FOR <CONTROL VAR> := <INITIAL VALUE> DOWNTO
(continue)                   <FINAL VALUE> <DO> <ST>
<CONTROL VAR> <VAR ID>
<INITIAL VALUE> <EXP>
<WHILE> WHILE
<DO> DO
<REPEAT> REPEAT
<UNTIL> UNTIL
<FINAL VALUE> <EXP>
<WITH ST> WITH <RECORD VAR LIST> DO <ST>
<RECORD VAR LIST> <RECORD VAR LIST> , <VAR>
                  <VAR>
<CONST> <UNSIGNED INTEGER>
        <UNSIGNED REAL>
        + <UNSIGNED INTEGER>
        - <UNSIGNED INTEGER>
        + <UNSIGNED REAL>
        - <UNSIGNED REAL>
        <CONST ID>
        + <CONST ID>
        - <CONST ID>
        <STRING>
```

Chapter 11

INTERMEDIATE CODE

11.1 ARCHITECTURE

The intermediate code is based on a hypothetical stack machine. Each instruction of this stack machine is composed of 3 parts with the following format.

```
OPCODE, L, A
```

The OPCODE is a mnemonic which specifies not only the operation but also the type(s) of the operand(s). L is the level difference between the current procedure and the procedure where the variable is defined. A is either a number (e.g. in LIT's, MARK), or a program address (e.g. in JUMP, CALL, etc.), or an offset of data address (e.g. in various load and store operations). A complete data address is composed of (L,A). Since an operand stack is used, there is no named register other than the program counter (PC), and it is not necessary to name explicitly the operand(s) of any arithmetic operation. Therefore, (L,A) is not used in arithmetic instructions (e.g. ADD, DIV, GTR, etc.), and it is set to (0,0). Following are some examples of translation from source program statements into intermediate code.

Example 1: The first statement of GCD program in the Pascal-M User Manual (Appendix A, p. 12)

```
X:=M;
```

where M has been declared as a constant 24, is translated into

```
LITB  0 24
STB   0  2
```

The first instruction means load constant 24 of type byte onto the operand stack. The second instruction means to store the top byte to location (0,2), which is the address of X.

Example 2: The statement X:=X-Y of the same program is translated into

```
LODB  0  2
LODB  0  3
SUBB  0  0
STB   0  2
```

The first instruction means load a byte onto the operand stack from location (0,2), which is X. The second instruction loads a byte from location (0,3), which is Y. The third instruction is a subtract operation; the 2 operands will be popped from the operand stack and the result will be pushed onto the stack. The last instruction stores the top byte of the operand stack into location (0,2).

Example 3: The following is a fragment of a Pascal-M program.

```
PROCEDURE:A;
  VAR X:BYTE;
  PROCEDURE:B;
    VAR Y:BYTE;
    BEGIN (* PROCEDURE B *)
         ...
         X:=X+Y;
         ...
    END (*PROCEDURE B *)
  BEGIN (* PROCEDURE A *)
    ...
```

Suppose X is translated to be at offset 2 of procedure A, and Y is translated to be at offset 3 of procedure B, then the statement X:=X+Y will be translated into

```
LODB  1  2
LODB  0  3
ADDB  0  0
STB   1  2
```

The major difference from example 2 is that X is translated into (1,2). This is because X is not a local variable, and X is declared in a procedure (A) which is one level farther out than the procedure (B) where the statement X:=X+Y appears.

11.2 SPECIFICATION

The following pages of this chapter attempt to specify the intermediate code in a more precise way -- by describing each intermediate-code instruction in Pascal. First we have to conceive the operand stack as having 8-bit width and unlimited depth. Two operations PUSH and POP are associated with accessing the operand stack: PUSH pushes one variable (of type byte) onto the stack, while POP pops the top element of the stack and stores it into the named variable.

In addition to PUSH and POP we need to define some meta-operations in order to specify fully what the intermediate code will do.

```
B_TO_I     change a type byte variable into type integer.
B_TO_R     change a type byte variable into type real.
I_TO_B     change a type integer variable into type byte.
R_TO_B     change a type real variable into type byte.
R_TO_I     change a type real variable into type integer.
MOD        residue; both operands and result are integers.
HBI        extract the higher byte of an integer.
LBI        extract the lower byte of an integer.
HBR        extract the higher byte of the fraction part
           of a real.
LBR        extract the lower byte of the fraction part
           of a real.
CHR        extract the characteristic part of a real.
FORMI      form an integer from 2 bytes.
FORMR      form a real from 3 bytes.
+, -, *, /, >, >=, <, <=, =
           these operations are self-evident.
```

Some variables and constants are used in describing the meaning of the intermediate code. They are declared as follows.

```
VAR LB,   (* lower byte of an integer or real     *)
    HB,   (* higher byte of an integer or real    *)
    LB2,  (* temporary LB                         *)
    HB2,  (* temporary HB                         *)
    CH,   (* characteristic part of a real        *)
    OP1B, (* 1st operand of a byte operation      *)
    OP2B  (* 2nd operand of a byte operation      *)
       : BYTE;
    TOP,  (* top addr+1 for last activ'n record   *)
    B,    (* base address of last activ'n record  *)
    OLDB, (* temp store for B                     *)
    PC,   (* program counter                      *)
    OP1I, (* 1st operand of an integer operation  *)
    OP2I  (* 2nd operand of an integer operation  *)
       : INTEGER;
    OP1R, (* 1st operand of a real operation      *)
```

```
    OP2R:(* 2nd operand of a real operation      *)
         REAL;
S:ARRAY[0..255] OF BYTE; (* main store for         *)
                         (* activation record.     *)
TEMP:ARRAY[1..8] OF BYTE;(* temporary store        *)
```

```
FUNCTION BASE(L:BYTE):INTEGER;
(* This function computes the base address of  *)
(* an activation record of level L.             *)
VAR ADDR: INTEGER; (* address of activ'n rec.   *)
BEGIN ADDR:=B; (* set current activ'n record    *)
               (* address to ADDR.              *)
      WHILE L>0 DO
       BEGIN L:=L-1;
             ADDR:=FORMI(S[ADDR],S[ADDR+1])
       END;
      BASE:=ADDR
END.
```

```
PROCEDURE REFBASE(VAR L:BYTE, VAR A:BYTE);
(* This procedure computes the relative address*)
(* (L,A) of a call by reference variable from  *)
(* the given indirect (L,A).                    *)
BEGIN PUSH(S[BASE(L)+A]+L+1);    (* store new L *)
      A:=S[BASE(L)+A+1];         (* get new A   *)
      L:=POP                     (* get new L   *)
END;
```

The following are the meanings of the 91 intermediate-code instructions.

```
(*  1 *)  'ADBI'    (* add byte to integer *)
                    LB:=POP;
                    HB:=POP;
                    OP1I:=FORMI(HB,LB);
                    OP2I:=B_TO_I(POP);
                    PUSH(HBI(OP1I+OP2I);
                    PUSH(LBI(OP1I+OP2I);

(*  2 *)  'ADDB'    (* add 2 bytes *)
                    OP1B:=POP;
                    OP2B:=POP;
                    PUSH(OP1B+OP2B);

(*  3 *)  'ADDI'    (* add 2 integer numbers *)
                    LB:=POP;
                    HB:=POP;
                    OP1I:=FORMI(HB,LB);
                    LB:=POP;
                    HB:=POP;
                    OP2I:=FORMI(HB,LB);
                    PUSH(HBI(OP1I+OP2I);
                    PUSH(LBI(OP1I+OP2I);

(*  4 *)  'ADDR'    (* add 2 real numbers *)
                    CH:=POP;
                    LB:=POP;
                    HB:=POP;
                    OP1R:=FORMR(CH,HB,LB);
                    CH:=POP;
                    LB:=POP;
                    HB:=POP;
                    OP2R:=FORMR(CH,HB,LB);
                    PUSH(HBR(OP2R+OP1R);
                    PUSH(LBR(OP2R+OP1R);
                    PUSH(CHR(OP2R+OP1R);

(*  5 *)  'ADIB'    (* add an integer with a byte *)
                    OP1I:=B_TO_I(POP);
                    LB:=POP;
                    HB:=POP;
                    OP2I:=FORMI(HB,LB);
                    PUSH(HBI(OP1I+OP2I);
                    PUSH(LBI(OP1I+OP2I);

(*  6 *)  'AND '    (* 'and' 2 bytes *)
                    OP1B:=POP;
                    OP2B:=POP;
                    IF OP1B=TRUE THEN PUSH(OP2B);
                                 ELSE PUSH(FALSE);
```

```
(*  7 *)  'CALL'    (* call subroutine *)
          (* For a procedure call, the semantic routine *)
          (* will take the following actions:           *)
          (*                                            *)
          (*  1. Push the actual parameters onto the    *)
          (*     operand stack.                         *)
          (*  2. Mark a new activation record and store *)
          (*     the dynamic link (DL) on it.           *)
          (*  3. Pop the actual parameters from the     *)
          (*     operand stack, and store them on the   *)
          (*     new activation record.                 *)
          (*  4. Prepare the static link (SL) and tran- *)
          (*     fer control to the new procedure.      *)
          (*                                            *)
          (* The I-code 'CALL' corresponds to the last  *)
          (* step of the semantic actions.              *)

                  IF A>0 THEN
                   BEGIN OLDB:=B;
                         B:=FORMI(S[B+2],S[B+3]);
                         IF L=0 THEN (* copy SL *)
                          BEGIN HB:=S[B];
                                LB:=S[B+1]
                          END
                         ELSE (* L=1, copy B to SL *)
                          BEGIN HB:=HBI(B);
                                LB:=LBI(B)
                          END;
                         B:=CLDB;
                         S[B]:=HB;   (* high byte of SL *)
                         S[B+1]:=LB; (* low byte of SL  *)
                         PUSH(LBI(PC));
                         PUSH(HBI(PC));
                         PC:=A   END
                  ELSE (* I/O procedures *)
                   CASE A OF
                         -7: READB;
                         -8: READI;
                         -9: READR;
                        -10: WRITEB;
                        -11: WRITEI;
                        -12: WRITER
                   END (* case *)
```

```
(*  8 *)   'CBRN'     (* change next_to_top from byte to *)
                      (* real.                           *)
                      LB:=POP;
                      HB:=POP;
                      CH:=POP;
                      OP2R:=B_TO_R(POP);
                      PUSH(CHR(OP2R);
                      PUSH(HBR(OP2R);
                      PUSH(LBR(OP2R);
                      PUSH(CH);
                      PUSH(HB);
                      PUSH(LB);

(*  9 *)   'CBRT'     (* change top from byte to real *)
                      OP1R:=B_TO_R(POP);
                      PUSH(HBR(OP1R);
                      PUSH(LBR(OP1R);
                      PUSH(CHR(OP1R);

(* 10 *)   'CIB '     (* change top from integer to byte *)
                      LB:=POP;
                      HB:=POP;
                      PUSH(LB);

(* 11 *)   'CIRN'     (* change next_to_top from integer *)
                      CH:=POP;          (* to real. *)
                      LB:=POP;
                      HB:=POP;
                      LB2:=POP;
                      HB2:=POP;
                      OP2R:=I_TO_R(FORMI(HB2,LB2));
                      PUSH(HBR(OP2R);
                      PUSH(LBR(OP2R);
                      PUSH(CHR(OP2R);
                      PUSH(HB);
                      PUSH(LB);
                      PUSH(CH);

(* 12 *)   'CIRT'     (* change top from integer to real *)
                      LB:=POP;
                      HB:=POP;
                      OP1R:=I_TO_R(FORMI(HB,LB));
                      PUSH(HBR(OP1R);
                      PUSH(LBR(OP1R);
                      PUSH(CHR(OP1R);

(* 13 *)   'COM '     (* 1's complement a byte *)
                      OP1B:=POP;
                      OP1B:=-1-OP1B;
                      PUSH(OP1B);
```

```
(* 14 *)   'CRBT'    (* change top from real to byte *)
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     OP1B:=R_TO_B(FORMR(CH,HB,LB));
                     PUSH(OP1B);

(* 15 *)   'CRIT'    (* change top from real to integer *)
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     OP1I:=R_TO_I(FORMR(CH,HB,LB));
                     PUSH(HBI(OP1I);
                     PUSH(LBI(OP1I);

(* 16 *)   'DIVB'    (* divide 2 bytes *)
                     OP1B:=POP;
                     OP2B:=POP;
                     PUSH(OP2B/OP1B);

(* 17 *)   'DIVI'    (* divide 2 integers *)
                     LB:=POP;
                     HB:=POP;
                     OP1I:=FORMI(HB,LB);
                     LB:=POP;
                     HB:=POP;
                     OP2I:=FORMI(HB,LB);
                     PUSH(HBI(OP2I/OP1I);
                     PUSH(LBI(OP2I/OP1I);

(* 18 *)   'DIVR'    (* divide 2 reals *)
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     OP1R:=FORMR(CH,HB,LB);
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     OP2R:=FORMR(CH,HB,LB);
                     PUSH(HBR(OP2R/OP1R);
                     PUSH(LBR(OP2R/OP1R);
                     PUSH(CHR(OP2R/OP1R);

(* 19 *)   'DVBI'    (* divide byte by integer *)
                     LB:=POP;
                     HB:=POP;
                     OP1I:=FORMI(HB,LB);
                     OP2I:=B_TO_I(POP);
                     PUSH(I_TO_B(OP2I/OP1I));
```

```
(* 20 *)   'DVIB'     (* divide integer by byte *)
                      OP1I:=B_TO_I(POP);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      PUSH(LBI(OP2I/OP1I);
                      PUSH(HBI(OP2I/OP1I);

(* 21 *)   'EQBI'     (* if a byte equals an integer? *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      OP2I:=B_TO_I(POP);
                      IF OP1I=OP2I THEN PUSH(TRUE);TE *)
                                   ELSE PUSH(FALSE);

(* 22 *)   'EQIB'     (* if an integer equals a byte? *)
                      OP1I:=B_TO_I(POP);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      IF OP1I=OP2I THEN PUSH(TRUE);
                                   ELSE PUSH(FALSE);

(* 23 *)   'EQUB'     (* if 2 bytes are equal? *)
                      OP1B:=POP;
                      OP2B:=POP;
                      IF OP1B=OP2B THEN PUSH(TRUE);
                                   ELSE PUSH(FALSE);

(* 24 *)   'EQUI'     (* if 2 integers are equal? *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      LB:=POP;
                      LB:=POP;
                      OP2I:=FORMI(HB,LB);
                      IF OP1I=OP2I THEN PUSH(TRUE);
                                   ELSE PUSH(FALSE);

(* 25 *)   'EQUR'     (* if 2 reals are equal? *)
                      CH:=POP;
                      LB:=POP;
                      HB:=POP;
                      OP1R:=FORMR(CH,HB,LB);
                      CH:=POP;
                      LB:=POP;
                      HB:=POP;
                      OP2R:=FORMR(CH,HB,LB);
                      IF OP1R=OP2R THEN PUSH(TRUE);
                                   ELSE PUSH(FALSE);
```

```
(* 26 *)    'GEB '     (* if next_to_top byte >= top byte? *)
                       OP1B:=POP;
                       OP2B:=POP;
                       IF OP2B>=OP1B THEN PUSH(TRUE);
                                     ELSE PUSH(FALSE);

(* 27 *)    'GEBI'     (* if next_to_top byte >= top  *)
                       (* integer?                    *)
                       LB:=POP;
                       HB:=POP;
                       OP1I:=FORMI(HB,LB);
                       OP2I:=B_TO_I(POP);
                       IF OP2I>=OP1I THEN PUSH(TRUE);

(* 28 *)    'GEI '     (* if next_to_top integer >= top *)
                       (* integer?                      *)
                       LB:=POP;
                       HB:=POP;
                       OP1I:=FORMI(HB,LB);
                       LB:=POP;
                       LB:=POP;
                       OP2I:=FORMI(HB,LB);
                       IF OP2I>=OP1I THEN PUSH(TRUE);
                                     ELSE PUSH(FALSE);

(* 29 *)    'GEIB'     (* if next_to_top integer >= top *)
                       (* byte?                         *)
                       OP1I:=B_TO_I(POP);
                       LB:=POP;
                       HB:=POP;
                       OP2I:=FORMI(HB,LB);
                       IF OP2I>=OP1I THEN PUSH(TRUE);
                                     ELSE PUSH(FALSE);

(* 30 *)    'GER '     (* if next_to_top real >= top real? *)
                       CH:=POP;
                       LB:=POP;
                       HB:=POP;
                       OP1R:=FORMR(CH,HB,LB);
                       CH:=POP;
                       LB:=POP;
                       HB:=POP;
                       OP2R:=FORMR(CH,HB,LB);
                       IF OP2R>=OP1R THEN PUSH(TRUE);
                                     ELSE PUSH(FALSE);

(* 31 *)    'GTB '     (* if next_to_top byte > top byte? *)
                       OP1B:=POP;
                       OP2B:=POP;
                       IF OP2B> OP1B THEN PUSH(TRUE);
                                     ELSE PUSH(FALSE);
```

```
(* 32 *)   'GTBI'     (* if next_to_top byte > top   *)
                      (* integer?                    *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      OP2I:=B_TO_I(POP);
                      IF OP2I>OP1I THEN PUSH(TRUE);

(* 33 *)   'GTI '     (* if next_to_top integer > top *)
                      (* integer?                     *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      LB:=POP;
                      LB:=POP;
                      OP2I:=FORMI(HB,LB);
                      IF OP2I> OP1I THEN PUSH(TRUE);
                                    ELSE PUSH(FALSE);

(* 34 *)   'GTIB'     (* if next_to_top integer > top *)
                      (* byte?                        *)
                      OP1I:=B_TO_I(POP);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      IF OP2I>OP1I THEN PUSH(TRUE);;
                                   ELSE PUSH(FALSE);

(* 35 *)   'GTR '     (* if next_to_top real > top real? *)
                      CH:=POP;
                      LB:=POP;
                      HB:=POP;
                      OP1R:=FORMR(CH,HB,LB);
                      CH:=POP;
                      LB:=POP;
                      HB:=POP;
                      OP2R:=FORMR(CH,HB,LB);
                      IF OP2R>OP1R THEN PUSH(TRUE);
                                   ELSE PUSH(FALSE);

(* 36 *)   'JPF '     (* jump false *)
                      OP1B:=POP;
                      IF OP1B=FALSE THEN PC:=A;

(* 37 *)   'JPT '     (* jump true *)
                      OP1B:=POP;
                      IF OP1B=TRUE THEN PC:=A;

(* 38 *)   'JUMP'     (* jump  *)
                      PC:=A;
```

```
        (* For loading array variables, the semantic   *)
        (* actions are:                                 *)
        (*                                              *)
        (*   1. Evaluate the subscripts.                *)
        (*   2. Convert the subscripts into an offset   *)
        (*      of an activation record.                *)
        (*   3. Leave the offset on the operand stack   *)
        (*      as it is being calculated.              *)
        (*   4. Get the offset from the operand stack   *)
        (*      and load the variable accordingly onto *)
        (*      the operand stack.                      *)
        (*                                              *)
        (* The following 3 LDX's correspond to the     *)
        (* last step of the above actions.              *)

(* 39 *)  'LDXB'    (* load byte indirect *)
                    LB:=POP;

(* 40 *)  'LDXI'    (* load integer indirect *)
                    LB:=POP;
                    PUSH(S[BASE(L)+LB]);    (* high byte *)
                    PUSH(S[BASE(L)+LB+1]);  (* low byte  *)

(* 41 *)  'LDXR'    (* load real indirect *)
                    LB:=POP;
                    PUSH(S[BASE(L)+LB]);    (* high byte *)
                    PUSH(S[BASE(L)+LB+1]);  (* low byte  *)
                    PUSH(S[BASE(L)+LB+2]);  (* exponent  *)

(* 42 *)  'LITB'    (* load literal byte *)
                    PUSH(A);

(* 43 *)  'LITI'    (* load literal integer *)
                    PUSH(HBI(A));
                    PUSH(LBI(A));

(* 44 *)  'LITR'    (* load literal real *)
                    PUSH(HBI(A));
                    PUSH(LBI(A));
                    PUSH(L); (* exponent *)

(* 45 *)  'LODB'    (* load byte *)
                    PUSH(S[BASE(L)+A]);

(* 46 *)  'LODI'    (* load integer *)
                    PUSH(S[BASE(L)+A]);   (* high byte *)
                    PUSH(S[BASE(L)+A+1]); (* low byte *)
```

```
(* 47 *)   'LODR'     (* load real *)
                      PUSH(S[BASE(L)+A]);   (* high byte *)
                      PUSH(S[BASE(L)+A+1]); (* low byte *)
                      PUSH(S[BASE(L)+A+2]); (* exponent *)

(* 48 *)   'LRFB'     (* load call by reference byte *)
                      REFBASE(L,A);
                      PUSH(S[BASE(L)+A]);

(* 49 *)   'LRFI'     (* load call by reference integer *)
                      REFBASE(L,A);
                      PUSH(S[BASE(L)+A]);   (* high byte *)
                      PUSH(S[BASE(L)+A+1]); (* low byte *)

(* 50 *)   'LRFR'     (* load call by reference real *)
                      REFBASE(L,A);
                      PUSH(S[BASE(L)+A]);   (* high byte *)
                      PUSH(S[BASE(L)+A+1]); (* low byte *)
                      PUSH(S[BASE(L)+A+2]); (* exponent *)

(* 51 *)   'LTPB'     (* load byte from temp area *)
                      PUSH(TEMP[A]);

(* 52 *)   'LTPI'     (* load integer from temp area *)
                      PUSH(TEMP[A]);   (* HB *)
                      PUSH(TEMP[A+1]); (* LB *)

(* 53 *)   'MARK'     (* mark new activation record        *)
           (* This instruction corresponds to the second *)
           (* step of semantic action for procedure      *)
           (* invocation.  See 'CALL' for details.       *)
                      S[TOP+2]:=HBI(B); (* DL *)
                      S[TOP+3]:=LBI(B); (* DL *)
                      B:=TOP;
                      TOP:=TOP+A;

(* 54 *)   'MDBI'     (* mod byte by integer *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      OP2I:=B_TO_I(POP);
                      PUSH(I_TO_B(MOD(OP2I,OP1I)));

(* 55 *)   'MDIB'     (* mod integer by byte *)
                      OP1I:=B_TO_I(POP);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      PUSH(HBI(MOD(OP2I,OP1I)));
                      PUSH(LBI(MOD(OP2I,OP1I)));
```

```
(* 56 *)   'MLBI'     (* multiply byte and integer *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      OP2I:=B_TO_I(POP);
                      PUSH(HBI(OP2I*OP1I));
                      PUSH(LBI(OP2I*OP1I));

(* 57 *)   'MLIB'     (* multiply byte and integer *)
                      OP1I:=B_TO_I(POP);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      PUSH(HBI(OP2I*OP1I));
                      PUSH(LBI(OP2I*OP1I));

(* 58 *)   'MODB'     (* mod 2 bytes *)
                      OP1B:=POP;
                      OP2B:=POP;
                      PUSH(MODB(OP2B,OP1B));

(* 59 *)   'MODI'     (* mod 2 integers *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      PUSH(HBI(MOD(OP2I,OP1I)));
                      PUSH(LBI(MOD(OP2I,OP1I)));

(* 60 *)   'MULB'     (* multiply 2 bytes *)
                      OP1B:=POP;
                      OP2B:=POP;
                      PUSH(OP1B*OP2B);

(* 61 *)   'MULI'     (* multiply 2 integers *)
                      LB:=POP;
                      HB:=POP;
                      OP1I:=FORMI(HB,LB);
                      LB:=POP;
                      HB:=POP;
                      OP2I:=FORMI(HB,LB);
                      PUSH(HBI(OP2I*OP1I);
                      PUSH(LBI(OP2I*OP1I);
```

```
(* 62 *)   'MULR'   (* multiply 2 reals *)
                    CH:=POP;
                    LB:=POP;
                    HB:=POP;
                    OP1R:=FORMR(CH,HB,LB);
                    CH:=POP;
                    LB:=POP;
                    HB:=POP;
                    OP2R:=FORMR(CH,HB,LB);
                    PUSH(HBR(OP2R*OP1R);
                    PUSH(LBR(OP2R*OP1R);
                    PUSH(CHR(OP2R*OP1R);

(* 63 *)   'NEGB'   (* negate a byte *)
                    OP1B:=POP;
                    PUSH(-OP1B);

(* 64 *)   'NEGI'   (* negate an integer *)
                    LB:=POP;
                    HB:=POP;
                    OP1I:=FORMI(HB,LB);
                    PUSH(HBI(-OP1I);
                    PUSH(LBI(-OP1I);

(* 65 *)   'NEGR'   (* negate a real *)
                    CH:=POP;
                    LB:=POP;
                    HB:=POP;
                    OP1R:=FORMR(CH,HB,LB);
                    PUSH(HBR(-OP1R);
                    PUSH(LBR(-OP1R);
                    PUSH(CHR(-OP1R);

(* 66 *)   'OR  '   (* 'or' 2 bytes *)
                    OP1B:=POP;
                    OP2B:=POP;
                    IF OP1B=FALSE THEN PUSH(OP2B);
                                  ELSE PUSH(TRUE);

(* 67 *)   'POP '   (* pop a byte *)
                    DO WHILE(A>0);
                       POP;
                       A:=A-1;
                    END;

(* 68 *)   'RTS '   (* return from subroutine *)
                    TOP:=B;
                    HB:=S[B+2];
                    LB:=S[B+3];
                    B:=FORMI(HB,LB);
                    PC:=FORMI(POP,POP);
```

```
(* 69 *)   'SBBI'    (* subtract integer from byte *)
                     LB:=POP;
                     HB:=PCP;
                     OP1I:=FORMI(HB,LB);
                     OP2I:=B_TO_I(POP);
                     PUSH(HBI(OP2I-OP1I);
                     PUSH(LBI(OP2I-OP1I);

(* 70 *)   'SBIB'    (* subtract byte from integer *)
                     OP1I:=B_TO_I(POP);
                     LB:=POP;
                     HB:=POP;
                     OP2I:=FORMI(HB,LB);
                     PUSH(HBI(OP2I-OP1I);
                     PUSH(LBI(OP2I-OP1I);

(* 71 *)   'SFBI'    (* store byte to call by ref integer *)
                     REFBASE(L,A);
                     LB:=POP;
                     IF LB<0 THEN S[BASE(L)+A]:='11111111'B;
                             ELSE S[BASE(L)+A]:=0;
                     S[BASE(L)+A+1]:=LB;

(* 72 *)   'SFIB'    (* store integer to call by ref byte *)
                     REFBASE(L,A);
                     S[BASE(L)+A]:=POP;
                     POP; (* discard HB *)

(* 73 *)   'SRFB'    (* store call by ref byte *)
                     REFBASE(L,A);
                     S[BASE(L)+A]:=POP;

(* 74 *)   'SRFI'    (* store call by ref integer *)
                     REFBASE(L,A);
                     S[BASE(L)+A+1]:=POP; (* low byte *)
                     S[BASE(L)+A]:=POP;   (* high byte *)

(* 75 *)   'SRFR'    (* store call by ref real *)
                     REFBASE(L,A);
                     S[BASE(L)+A+2]:=POP; (* exponent *)
                     S[BASE(L)+A+1]:=POP; (* low byte *)
                     S[BASE(L)+A]:=POP;   (* high byte *)

(* 76 *)   'STB '    (* store byte to activation record *)
                     S[BASE(L)+A]:=POP;

(* 77 *)   'STBI'    (* store byte to integer *)
                     LB:=POP;
                     IF LB<0 THEN S[BASE(L)+A]:='11111111'B;
                             ELSE S[BASE(L)+A]:=0;
                     S[BASE(L)+A+1]:=LB;
```

```
(* 78 *)   'STI '    (* store integer *)
                     S[BASE(L)+A+1]:=POP; (* low byte *)
                     S[BASE(L)+A]:=POP;   (* high byte *)

(* 79 *)   'STIB'    (* store integer to byte *)
                     S[BASE(L)+A]:=POP;
                     POP;

(* 80 *)   'STPB'    (* store byte to temp area *)
                     TEMP[A]:=POP;

(* 81 *)   'STPI'    (* store integer to temp area *)
                     TEMP[A+1]:=POP;  (* LB *)
                     TEMP[A]:=POP;    (* HB *)

(* 82 *)   'STR '    (* store real *)
                     S[BASE(L)+A+2]:=POP; (* exponent *)
                     S[BASE(L)+A+1]:=POP; (* low byte *)
                     S[BASE(L)+A]:=POP;   (* high byte *)

(* 83 *)   'SUBB'    (* subtract 2 bytes *)
                     OP1B:=POP;
                     OP2B:=POP;
                     PUSH(OP2B-OP1B);

(* 84 *)   'SUBI'    (* subtract 2 integers *)
                     LB:=POP;
                     HB:=POP;
                     OP1I:=FORMI(HB,LB);
                     LB:=POP;
                     HB:=POP;
                     OP2I:=FORMI(HB,LB);
                     PUSH(HBI(OP2I-OP1I);
                     PUSH(LBI(OP2I-OP1I);

(* 85 *)   'SUBR'    (* subtract 2 reals *)
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     OP1R:=FORMR(CH,HB,LB);
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     OP2R:=FORMR(CH,HB,LB);
                     PUSH(HBR(OP2R-OP1R);
                     PUSH(LBR(OP2R-OP1R);
                     PUSH(CHR(OP2R-OP1R);
```

```
(* 86 *)   'SXB '    (* store indirect byte *)
                     LB:=POP;
                     S[BASE(L)+POP]:=LB;

(* 87 *)   'SXBI'    (* store indirect byte to integer *)
                     LB:=POP;
                     IF LB<0 THEN HB:='11111111'B;
                             ELSE HB:=0;
                     A:=POP;
                     S[BASE(L)+A]:=HB;
                     S[BASE(L)+A+1]:=LB;

(* 88 *)   'SXI '    (* store indirect integer *)
                     LB:=POP;
                     HB:=POP;
                     A:=POP;
                     S[BASE(L)+A]:=HB;
                     S[BASE(L)+A+1]:=LB;

(* 89 *)   'SXIB'    (* store indirect integer to byte *)
                     LB:=POP;
                     HB:=POP;
                     A:=POP;
                     S[BASE(L)+A]:=LB;

(* 90 *)   'SXR '    (* store indirect real *)
                     CH:=POP;
                     LB:=POP;
                     HB:=POP;
                     A:=POP;
                     S[BASE(L)+A]:=HB;
                     S[BASE(L)+A+1]:=LB;
                     S[BASE(L)+A+2]:=CH;

(* 91 *)   'X   '    (* do nothing; this code is a result *)
                     (* of optimization.                  *)
```

Chapter 12

CODE GENERATION

The procedure GENCODE performs the following tasks:

1. Optimize the intermediate code by

   a. eliminating load-store pairs.
   b. partially unfolding some code sequences that involve constants.
   c. eliminating indirect jumps.

2. Generate object code for the MC6800.

It is invoked only when the return code of the first phase of compilation is zero. It has 4 short internal procedures:

1. GENM: appends new code to the array MCODE.

2. HEX: converts a decimal number into 2 hexadecimal digits.

3. LOADA: generates optimized code for machine instruction 'LDAA #IMMED' (load an immediate operand into accumulator A) by testing if IMMED is zero or not; if so, then generates CLRA (clear accumulate A) instead of a load instruction.

4. LEVOFF, generates optimized code for

   ```
   LDAA  #LEV(I)
   LDAB  #OFF(I)
   ```

   by using the same method as for LOADA.

12.1 MEMORY ORGANIZATION

The memory of the object machine is divided into 5 parts:

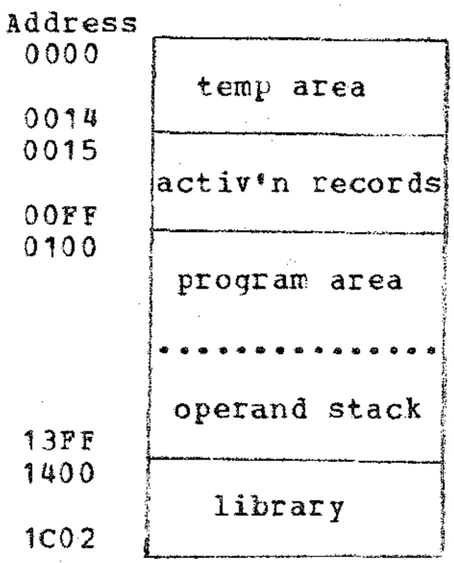
1. Addresses 0000--0014: temporary storage area, reserved for temporary variables and some static variables.

2. Addresses 0015--00FF: activation record area. The organization of an activation record is discussed later.

3. Addresses 0100--????: program area. This area extends toward the operand stack, but does not overlap it.

4. Addresses ????--13FF: the operand stack, with its bottom at address 13FF.

5. Addresses 1400--1BFF: the run time library. All the routines like multiplication, division, and all the routines for household chores are stored in this area. They are listed in section 12.2.



Memory organization

The activation record is organized as follows: memory locations 000C and 000D are for the variable B, which is the pointer to the base address of the current activation record; memory locations 000E and 000F are for the variable TOP, which points to the top of current activation record. In each activation record (except the first), the first byte is the lower byte of the address of static link (since the activation records are located at page zero, the higher byte for the address of any activation record is 00), the second byte is the lower byte of the address of dynamic link, and the rest of the bytes are for parameters passed to this procedure or local variables. Note that the return address is not on the activation record; it is on the hardware linkage stack of the machine. The following graph is a snap shot for activation records of the factorial program (see Sect. 5.6, Pascal-M User Manual) when N is 3 and the procedure FACTOF is just being invoked for the second time.

Address		
000C	00	B
000D	1F	
000E	00	TOP
000F	26	
0015	03	N
0016	00	F
0017	00	
0018	15	SL
0019	15	DL
001A	03	N
001B	00	addr.of F
001C	01	
001D	00	ICCAL_F
001E	00	
001F	15	SL
0020	18	DL
0021	02	N
0022	00	addr. of F
0023	05	
0024	00	LCCAL_F
0025	00	

12.2 RUN TIME LIBRARY

Except for the most trivial ones, every intermediate code instruction has a corresponding run time routine stored in the library, which shortens the program itself considerably. The following is the list of run time routines and their address and function.

routine	address	function
GBASE	1400	get the address of (L,A) into index register.
LODB	141E	routine for I-code LODB
LODI	142C	routine for I-code LODI
LODR	143F	routine for I-code LODR
STB	1459	routine for I-code STB
STI	146D	routine for I-code STI
STR	1488	routine for I-code STR
STBI	14AA	routine for I-code STBI
STIB	14C5	routine for I-code STIB
MARK	14DA	routine for I-code MARK
CALL0	14EE	routine for I-code CALL if level difference between caller and called is 0.
CALL1	1503	routine for I-code CALL if level difference between caller and called is 1.
REF	1510	get the base address for call by reference variables.
LRFB	1528	routine for I-code LRFB
LRFI	1535	routine for I-code LRFI
LRFR	153F	routine for I-code LRFR
SFBI	1549	routine for I-code SFBI
SFIB	1553	routine for I-code SFIB
SRFB	155D	routine for I-code SRFB
SRFI	1567	routine for I-code SRFI
SRFR	1571	routine for I-code SRFR
EQUB	157E	routine for I-code EQUB
WRITEI	158B	routine for standard procedure WRITEI
GTB	15AA	routine for I-code GTB
GEB	15B6	routine for I-code GEB
DIVIB	183B	routine for I-code DVIB
DIVI	1847	routine for I-code DIVI
MODIB	1852	routine for I-code MDIB
MODI	185E	routine for I-code MODI
DIVR	184A	routine for I-code DIVR
MULR	192F	routine for I-code MULR
MULI	196F	routine for I-code MULI
ADDBI	1990	routine for I-code ADBI
SUBBI	199C	routine for I-code SBBI
ADDIB	19A8	routine for I-code ADIB
SUBIB	19B4	routine for I-code SBIB
ADDI	19C6	routine for I-code ADDI
SUBI	19D0	routine for I-code SUBI
CRB	19DF	routine for I-code CRB
CRI	19E7	routine for I-code CRI
CIRN	1A1D	routine for I-code CIRN
CBRN	1A2F	routine for I-code CBRN
CIRT	1A44	routine for I-code CIRT
CBRT	1A5D	routine for I-code CBRT
SUBR	1A6E	routine for I-code SUBR

ADDR	1A74	routine for I-code ADDR
DIVB	1AED	routine for I-code DIVB
MODB	1AF0	routine for I-code MODB
DIVBI	1AF9	routine for I-code DVBI
MODBI	1AFC	routine for I-code MDBI
MULB	1B62	routine for I-code MULB
MULBI	1B75	routine for I-code MLBI
MULIB	1B84	routine for I-code MLIB
WRITEB	1BC4	routine for standard procedure WRITEB
READB	1BD6	routine for standard procedure READB

BIBLIOGRAPHY

[1] Aho, A.V., and Johnson, S.C., LR Parsing
Computing Surveys, Vol.6, No.2, June 1974, pp. 99-124.

[2] Aho, A.V., and Ullman, J.D., Principles of Compiler Design
Reading, MA: Addison-Wesley, 1977.

[3] Alpert, S.R., Pascal: A Structurally Strong Language
Peterborough, NH: Byte Publications Inc., Byte Magazine, Vol.3, No.8, August 1978, pp. 78-88.

[4] Ammann, U., et al., The Pascal (P) Compiler : Implementation Notes
revised edition. Zurich: Eidgenossische Technische Hochschule, July 1976.

[5] Bell, J.R., Threaded Code
Communications of the ACM, Vol.16 No.6, June 1973, pp. 370-372.

[6] Blaauw, G.A. Digital System Implementation
Englewood Cliffs, NJ: Prentice-Hall, 1976.

[7] Brooks, F.P., Jr., An Overview of Microcomputer Architecture and Software
Second Symposium on Micro Architecture, EUROMICRO, 1976.

[8] Calingaert, P., Assemblers, Compilers, and Program Translation
Potomac, MD: Computer Science Press, Inc. 1978.

[9] Conway, R., Gries, D., and Zimmerman, E.C.
A Primer on PASCAL
Cambridge, MA: Winthrop, 1976.

[10] DeRemer, F.L., Simple LR(K) Grammars
Communications of the ACM, Vol.14, No.7, July 1971, pp. 453-460.

[11] Dreizen, H.M., Micro-C User's Manual
Chicago: Dept. of Information Engineering, University of Illinois at Chicago Circle, 1976.

[12] Fuchs, H., et al., A System for Automatic Acquisition of Three Dimensional Data
Proc. of National Computer Conference, 1977, pp. 49-53

[13] Gries, D., _Compiler Construction for Digital Computers_
New York: John Wiley & Sons, 1971.

[14] Intel Corp, _8008 and 8080 PL/M programming manual_
Santa Clara, CA: Intel Corp, 1975.

[15] Jensen, K., and Wirth, N., _Pascal -- User Manual and Report_
2nd edition, New York, Berlin: Springer-Verlag, 1975.

[16] Marcotty, M., Ledgard, H.F., Bochmann, G.V., A sampler of Formal Definitions
Computing Surveys, Vol. 8, No. 2, June 1976, pp. 191-276.

[17] _Microprocessor Application Manual_
Motorola Semiconductor Products Inc.
New York: McGraw-Hill,1975.

[18] Organick, E.I., _Computer System Organization: The B5700/6700 Series_
New York: Academic Press, Inc., 1973.

[19] Ritter, T., and Boney, J., A Microprocessor For the Revolution: the 6809
Peterborough, NH: Byte Publications Inc., Byte Magazine, Vol.4, No.1, Jan. 1979, pp. 14-42.

[20] _Smoke Signal Broadcasting, BFD-68 System Manual_
Hollywood, CA: Smoke Signal Broadcasting, 1977.

[21] _SWTBUG, 6800 ROM Monitor, version 1.0, Users Guide_
San Antonio, TX: Southwest Technical Prods. Corp, 1977.

[22] Wirth, N., _Algorithms + Data Structures = Programs_
Englewood Cliffs, NJ: Prentice-Hall, 1976.