186

PASCAL/S:
SEQUENTIAL PASCAL
WITH DATA TYPE EXTENSIONS

BY

BARBARA K. NORTH

B.S., KANSAS STATE UNIVERSITY, 1970
M.S., KANSAS STATE UNIVERSITY, 1972

3

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1978

Approved by:

Virg Wallentine

Major Professor

ACKNOWLEDGEMENTS

The writer is indebted to many persons whose assistance made possible the completion of this paper.

Much appreciation is due Gary G. Anderson, the project coordinator. His exceptional talents, intellect, and enthusiasm were most valuable in the development and completion of this work. As a mentor and friend he is a rare and inspirational example.

For his encouragement and confidence, much appreciation is also due Virgil Wallentine, the writer's major professor.

The personal and professional contributions of Fred Maryanski and Beth Unger throughout the writer's program were greatly appreciated also.

The writer is greatful for the support of friends and colleagues in the department. Much thanks is also due the faculty in Computer Science for the facilities and opportunities which are provided. Madi McArthur deserves special thanks for her patience and artistic talents.

The writer is especially humbled by the special contribution of her husband, Skip, whose personal sacrifices made possible the completion of this report.

For their interest and financial support, John Goettelmann and Perkin-Elmer Data Systems deserve much thanks.

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

1.1: Description of the Project

This report describes a ten pass optimizing compiler for the programming language Sequential Pascal (with extensions). This version of the compiler was developed by Gary G. Anderson in a supported research activity through the Department of Computer Science and Kansas State University. (This research effort was funded in total by a grant from Perkin-Elmer Data Systems.)

The PASCAL/S compiler documented herein is an extension of the Sequential Pascal compiler, which was adapted from the Concurrent Pascal Compiler developed by Hartmann. [Hart75a]

The language Pascal was originally proposed by Niklaus Wirth and is described in detail in the following reports:

Wirth, N., Systematic Programming, Prentice-Hall, 1973.

Jensen, K. and Wirth, N., Pascal - User Manual and Report, Lecture Notes in Computer Science 18, Springer-Verlag, 1974.

This report is written for an audience already somewhat familiar with the language Sequential Pascal and the Hartmann SPASCAL compiler. Those readers seeking a better foundation in the language of Pascal are referred to the following sources:

Hankley W. and Rawlinson, J. _Sequential Pascal Supplement for Fortran Programmers: A Primer of Slides_, Computer Science Department, Kansas State University, 1977.

Brinch Hansen, P. _Sequential Pascal Report_, Information Science, California Institute of Technology, July,1975.

The PASCAL/S version of the Sequential Pascal Compiler is currently running on an Interdata 8/32 at Kansas State University. As with previous versions, this compiler is written is Sequential Pascal making it easily portable to other machines. The code generated is interpreted by a virtual machine which is then simulated on the real machine, in the current implementation at Kansas State. However, the code generated by the compiler is designed to match the instruction set of a prototype machine currently being develcped by Perkin-Elmer Data Systems.

The main focus of this report is to document the PASCAL/S Compiler. However, this report along with the accompanying appendices may serve as a tutorial for those learning Sequential Pascal for use with this compiler.

The remainder of this chapter includes an introduction to the Sequential Pascal compiler implemented by Hartmann, an introduction to the PASCAL/S compiler, and a descripton of the remainder of the report.

1.2: JUSTIFICATION FOR PASCAL

The compiler for Brinch Hansen's Concurrent pascal programming language [Brin75a] was developed by Hartmann in his dissertation [Hart75a]. It delineates seven sequential passes and provides the foundation for the current work. Hartmann's compiler for Sequential Pascal was adapted from the concurrent compiler he documented.

Hartmann's Sequential Pascal compiler was implemented at Kansas State University in October, 1976 on an Interdata 8/32. Simultaneously, the Navy Ocean Systems Center (formerly the Navy Undersea Center) [Ball 76a] was working on its implementation for an Interdata 7/16. Following these implementations several modifications were made to the compiler by each group. While several of these changes are reflected in the current version, it is primarily Hartmann's compiler for Sequential Pascal which provided the fundamental structure for development of the PASCAL/S compiler.

Pascal, in particular Hartmann's compiler, was selected for use at Kansas State in the development of the compiler for PASCAL/S hardware for the following reasons:

Availability : A version of Sequential Pascal

(SPASCAL) was available on the Interdata 8/32 at Kansas State for use by the research team.

Acceptability: Pascal, as currently implemented runs on a Pascal virtual machine which simulates a hardware stack machine. The idea of building a real machine close to the simulated stack machine makes Pascal a natural, logical choice for both the implementation language and for a basic compiler for the machine.

Ease of Modification: Since the compiler was itself written in a high level language, it is relatively easy to bootstrap up to a compiler for a new machine using this compiler. The fact that Hartmann's version of the compiler was written in Sequential Pascal and incorporated many features essential to the design of good software, means that fewer person-hours would be required to incorporate the desired changes.

Documentation: Pascal has been around for several years since some of the early work by Wirth [Wirt71a]. However its recent popularity is triggered by current publicity and many articles describing facets of the language. Most helpful among the recent articles are Per Brinch Hansen's Reports on Sequential [Brin75f] and Concurrent [Brin75c] Pascal and Alan Hartmann's Dissertation

[Hart75a]. These reports in conjunction with the source code for Hartmann's compiler certainly provided adequate documentation for beginning the project.

1.3: MODELS AND COMPILER DESIGN TECHNIQUES

Examining a simplistic graphical model of a compiler one could consider the components as a lexical box, a syntax box and a code generator, all sharing common tables. This naive model was extended by Gries [Grie71a]. Pictorially (See figure 1.) his model shows in detail the analysis of the source program (scanning and syntax/semantic analysis) and then synthesis of the object code (preparation for code generation and code generation). These simplistic models have their merit as learning models; however it is not always possible, nor desirable to implement these models in real compilers.

Hartmann described the compilation process as consisting of lexical analysis, syntax analysis, semantic analysis and code assembly. (See figure 2.) He used these divisions as a guideline to the pass structure. Semantic analysis was then refined further into three functional passes and code assembly was accomplished in the classic two pass fashion of code generation. Hartmann's compiler was modeled after the Gier Algol 68 compiler [Grie71a], which is the best known of the many-pass compilers.

Most of the techniques of compiler design used by Hartmann in the development of his compiler are well known.

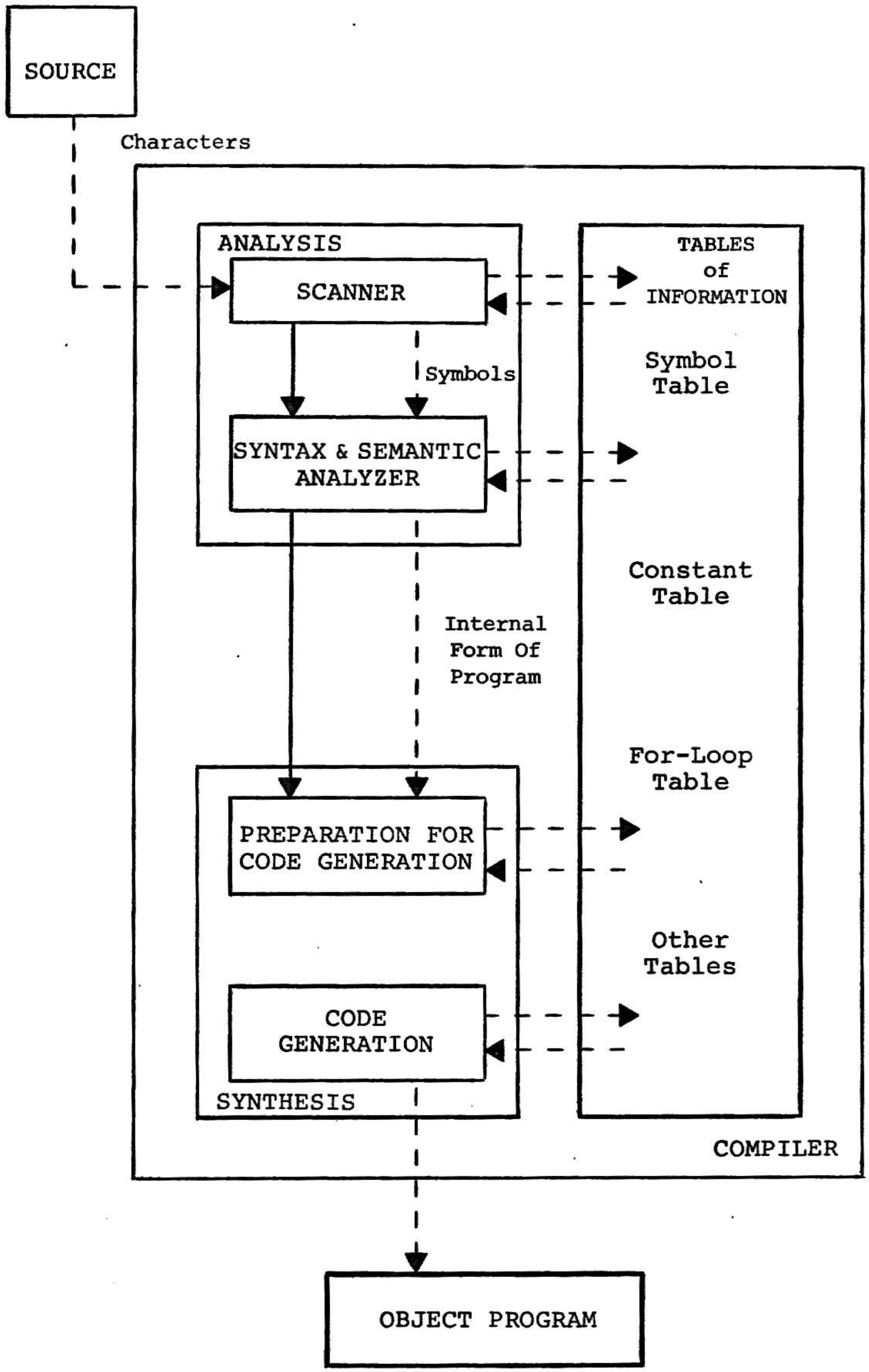


FIGURE 1: GRIES MODEL: LOGICAL PARTS OF THE COMPILER

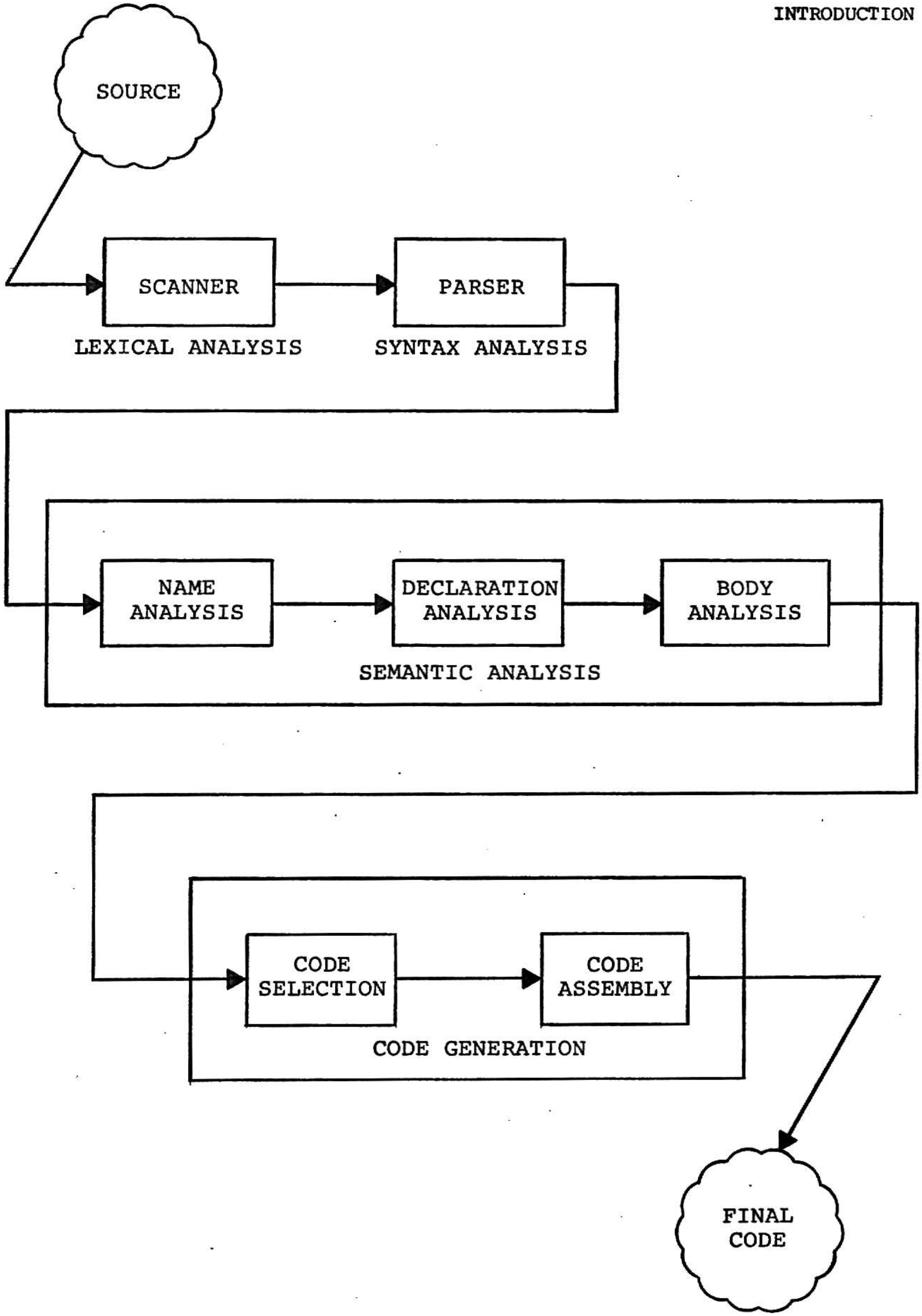


FIGURE 2: HARTMAN PASCAL COMPILER MODEL

For a many-pass compiler written in a high-level language, the Hartmann compiler has incorporated many features of good software design. That is, the compiler is quite modular as each construct within the compiler design comprises a separate pass. Briefly, these constructs and the techniques employed are described in the succeeding paragraphs. Figure 3 shows the data flow of the passes and the tables which are created and used in each of the passes. The tables and data structures depicted in figure 3 will be described within the discussion pertaining to each of the passes.

The **Lexical Analysis** performed in pass 1 uses a **finite state automata** to analyze the characters of the source language one character at a time. As such the characters are collected into groups which have a logical relationship called tokens. These tokens are representative of numeric constants, keywords (DO, IF, etc.) ,operator symbols (<=, +, *, etc.) or punctuation symbols (;, :, etc.). The output is a stream of integers which comprise the internal representations of these tokens. Gries [Grie71a] stresses reasons for treating the scanning or lexical analysis as a separate pass. Enumerated, these reasons are:

> 1. A larger portion of compile-time is spent in scanning characters. Separation allows us to concentrate on reducing this time. ---
> 2. The syntax can be described by very simple grammars. If we separate scanning from syntax recognition, we can develop efficient parsing techniques which are particularly well suited for these grammars. ---

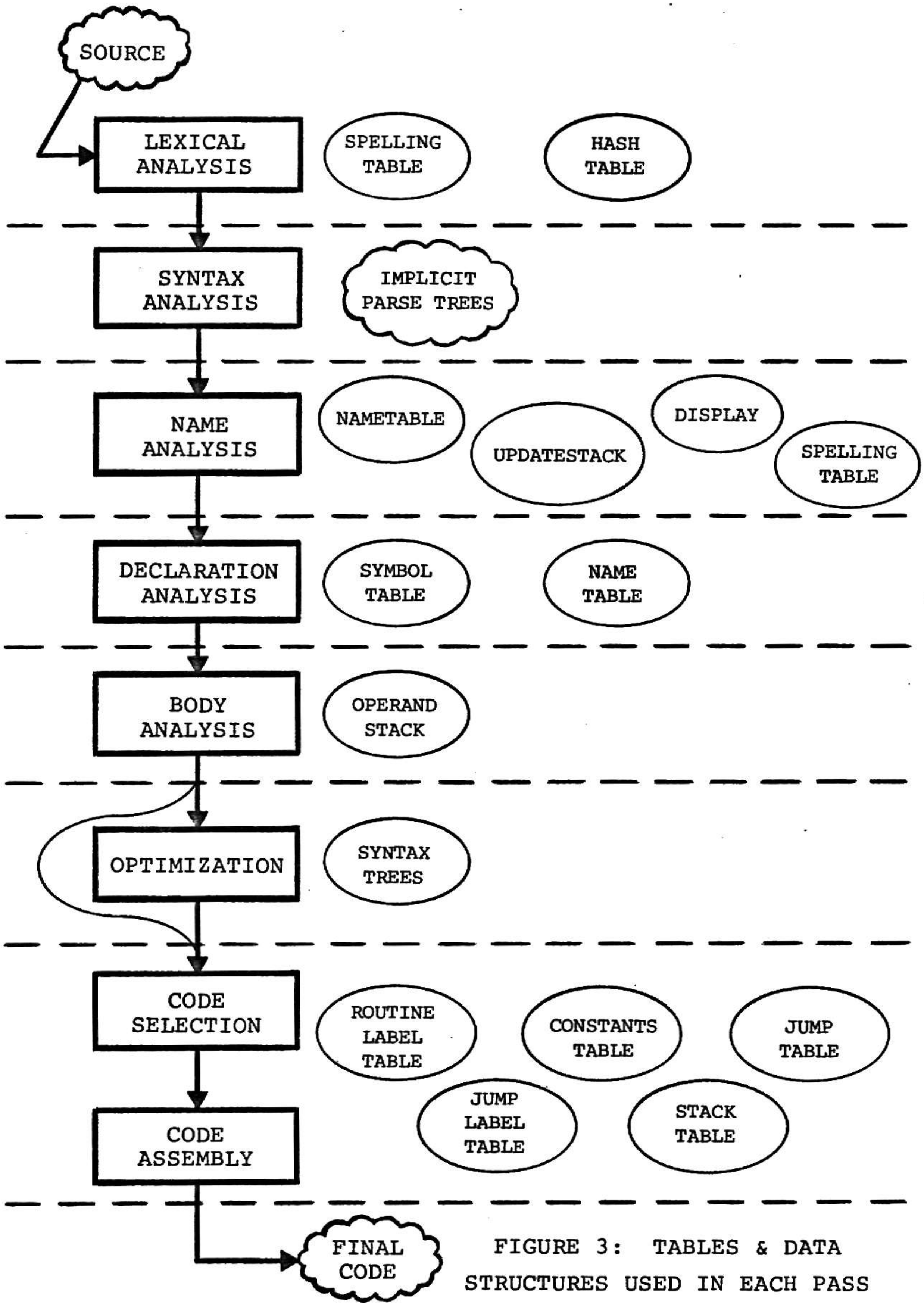


FIGURE 3: TABLES & DATA STRUCTURES USED IN EACH PASS

3. Since the scanner returns a symbol instead of a character, the syntax analyzer actually gets more information about what to do at each step.
---
4. Development of high-level languages requires attention to both lexical and syntactic properties. Separation of the two allows us to investigate them independently.
5. Often one has two different hardware representations for the same language. --- Separation allows us to write one syntactic analyzer and several scanners which are more simple and easier to write) ---one for each source program representation and/or input device.

The **Syntax analysis** performed in pass 2 uses a **top-down recursive descent parser.** There are two advantages for using this approach. The first is that the language's syntax is structured so that the parser needs no backtracking to recognize its input, since a set of recursive procedures is used to recognize the input. Secondly, this method provides flexibility in its ability to insert semantic constraints which will aid semantic analysis. However, this method may require more work in development than some other methods and probably is not the most efficient parsing technique. The recursive procedures contribute to structured programming and are, in general, fairly easy and efficient to write. The basic function of this pass is to insure that tokens from the input occur in patterns consistent with those specified for the language's syntax.

The semantic analysis performed in passes 3, 4 and 5 uses numerous data structures. The **Scope analysis or name analysis** done in pass 3 utilizes a name table, a modified compile-time display and several other data structures which are presented in more detail in the description of Hartmann's compiler. It is in this pass that unique spelling indices are transformed into unique name indices.

The **declaration analysis** accomplished by pass 4 uses a name table and symbol table. The **body analysis** performed in pass 5 also uses an operand stack with detailed entries. The specific implementations of these data structures are described in more detail in the introduction to Hartmann's compiler (Chapter 1, section 5) and the implementation of the PASCAL/S compiler (Chapter 2, section 2).

The **code generation** accomplished in pass 6 and pass 7 is developed in the classic two-pass design. The first of these two passes does code selection. Essentially, this pass defines the addresses of program labels, determines stack requirements of routines, constructs the constant table, and translates the input code into final code. This pass creates four tables, (a routine label table, a jump label table, a stack table, and a constants table) which are saved in the heap and used in the next pass. For more detail on the uses and implementation of these data structures, refer to the description of these passes in

Chapter 4.

The final pass makes use of those tables mentioned above, replacing labels by addresses, inserting exact stack lengths and writing error messages. More information relating to this pass is also provided in Chapter 4.

A Pascal program which is to be executed must be compiled and then interpreted. (See figure 4.)

There are some real restrictions when running under the Solo operating system developed by Brinch Hansen [Brin75g] with respect to 16-bit addresses. There is a size restriction on programs which can be successfully compiled, because of limited addressable memory. This in turn restricts the size of the heap which limits the size of the spelling table especially in pass 3 where the name table is dynamically allocated in the heap. With 16-bit addresses that portion of the operating system used by a sequential program (the process and its monitors) and any sequential program are limited to a maximum of 64K bytes.

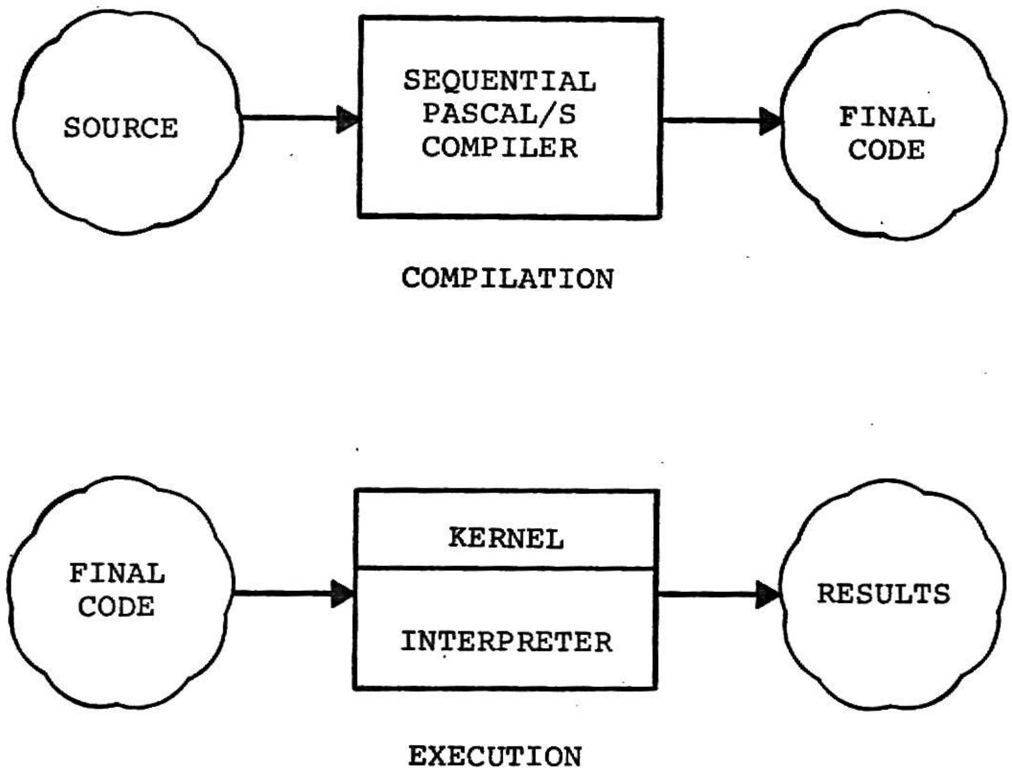


FIGURE 4: COMPILATION & EXECUTION

1.4: Definitions

Terms whose definitions are essential to the development of many of the notions within this paper are enumerated here for benefit of the reader. Many of these definitions are pertinent for the PASCAL/S version of the compiler and the Pascal language only.

source language: Basically the compiler must accept programs written in sequential pascal and translate them into the functionally equivalent machine language.

target language
or
object language: The final, expected code resulting from compilation.

compiler: Program which translates source language programs into object language. Compilers are essential to the execution process of a program written in a high-level language.

multi-pass compiler: This compiler requires several passes over the code to complete the translation from source language to target language. Thus, the translation is performed in degrees. The first pass maps the source code into the first intermediate language, the second into the second intermediate language, etc.

intermediate language: The language which is produced by all passes in a multi-pass compiler except the last pass which yields the target code.

source text: An instance of a source program consisting of a file of characters which are representative of a Sequential Pascal program.

intermediate code: The intermediate versions of the code resulting from an instance of a source program which consists of a file of integers. Each integer is representative of either an operator or an argument (operand) of an operator.

final code: The code of the target program.

program: A sequential Pascal program consists of a prefix, declarations and a body.

prefix: The prefix consists of constant, type and routine definitions which define the program's interface to the operating system [Neal77a].

declarations: The declarations assign names to constants, types, variables and routines.

body: Body implies that portion of the program which actually contains the statements to be executed by the machine.

machine: The final code consists of instructions for an SPASCAL/S machine. The machine is comprised of program space and data space [Goet77a].

program space: That portion of the machine which contains the target code.

data space: That portion of the machine which contains the program's variables and temporaries (including dynamic links).

driver: A control program which directs the flow of control cf calls to passes of the compiler.

parse tree: A diagram which exhibits the syntactic structure of an expression.

Syntax graphs have been adopted for use, by Brinch Hansen and Hartmann in describing the languages (source, intermediate, and target) of Pascal. Since they provide such a convenient way of specifying the languages associated with the compilation process, they have been adopted in this paper. Syntax graphs are directed graphs with terminals and /or non-terminals as the nodes. Syntax graphs completely define the syntactic specifications of the languages. All terminals or operators are shown in capital letters and they may be followed by arguments (operands) in parentheses. All non-terminals are defined by other syntax graphs.

Below are examples of syntax graphs defining the source language:

```
IDENTIFIER LIST

----------> ID ---------->
   |                 |
   |                 |
   <------ . <------


ID

----> LETTER -------------------->
               |                 |
               |<--LETTER <---|
               |<---DIGIT <---|
               |<---- . <-----|
```

'Identifier list' is a non-terminal defined by a syntax graph in which 'id' is another non-terminal defined by another syntax graph. A complete set of syntax graphs for the PASCAL/S source code, intermediate code and final code is included in Appendix B.

1.5: INTRODUCTION TO HARTMANN'S COMPILER

As stated previously, Hartmann's compiler includes one pass each for lexical analysis and syntax analysis, three passes for semantic analysis and two passes for code generation. These passes are identified as follows:

1. lexical analysis
2. syntax analysis
3. name analysis
4. declaration analysis
5. body analysis
6. code selection
7. code assembly

This multi-pass approach results in eight languages: the source language, the six intermediate languages and the target language. Hartmann's approach was to define the source first, the target second, and then to define the intermediate languages in the reverse order, starting with the last and ending with the first.

The syntax graphs of Wirth will be used to specify the intermediate languages. (See Appendix C). In the paragraphs that follow each of the passes will be described briefly.

Lexical Analysis converts the source text character by character into a sequence of integers representing operators, identifiers and constants. The integer representation of a unique identifier is a unique spelling index. This conversion yields the first intermediate code.

Syntax Analysis parses the program thereby checking the syntax of the first intermediate code. The resulting intermediate code is syntactically correct. This intermediate code is meaningful only to the extent that the input code was correct. The intermediate code produced is in postfix notation (operands followed by operators). Syntax analysis also replaces ambiguous operators by unique ones and eliminates redundant operators.

Name Analysis resolves any ambiguity in the use of identifiers by creating unique name indices from the spelling indices while enforcing the scope rules. Since Pascal allows the same identifier to refer to different types, variables or constants in different blocks, application of the scope rules is essential.

Declaration Analysis enforces the semantic rules of all declarations. Virtual addresses are assigned to routines and variables, and types are analyzed. This information is then distributed in line in the body of the program. With this information in the body where required, a simple design can be used for body analysis.

Body Analysis, the final phase of semantic processing, does the semantic checking of the body parts of the program. It checks the compatibility of operand types and their operators, resolves ambiguity, generates addressing commands for the machine, and distributes these commands in the body. The output from this pass is almost ready for the machine.

Code Selection is the first of the classic two-pass design used to perform code generation. Primarily it defines the addresses of program labels, determines stack requirements of routines, constructs the constants table and translates the input code into final code. This pass creates four tables (routine label table, a jump table, a stack table, and a constants table) which are saved in the heap and used in the next pass. "Code selection performs simple encoding of types into opcodes to make simulation of the virtual machine faster" according to Hartmann [Hart75a].

Code Assembly, the final pass, replaces program labels by addresses saved in the tables created in the previous pass, stack lengths are inserted, and error messages are listed.

1.6: INTRODUCTION TO THE PASCAL/S VERSION

The PASCAL/S compilers for Concurrent and Sequential Pascal were developed for an architecture under design by Perkin-Elmer Data Systems. Each of these compilers is comprised of ten passes. The first five passes (lexical analysis, syntax analysis, and three semantic passes) are quite similar to those of the Hartmann compiler described in the preceding section. Extensions and modifications to Hartmann's first five passes will be discussed in detail in Chapter 2. Of the five remaining passes in the PASCAL/S compiler, the next three (constant folding, expression evaluation, and ad hoc optimization) are optimization passes and the subsequent two passes are for code generation (code selection and code assembly).

All of the optimizing passes are completely optional. Each of the optimization passes will be described briefly in the following paragraphs and they will be followed by a discussion of the code generation passes. (See figure 5.)

Constant Folding is the first of the optimizing passes. This pass does at compile time all the operations which are static rather than leaving them to be done at run-time. With respect to operators, arithmetic on constants is done. Some Boolean expressions can also be eliminated at compile-

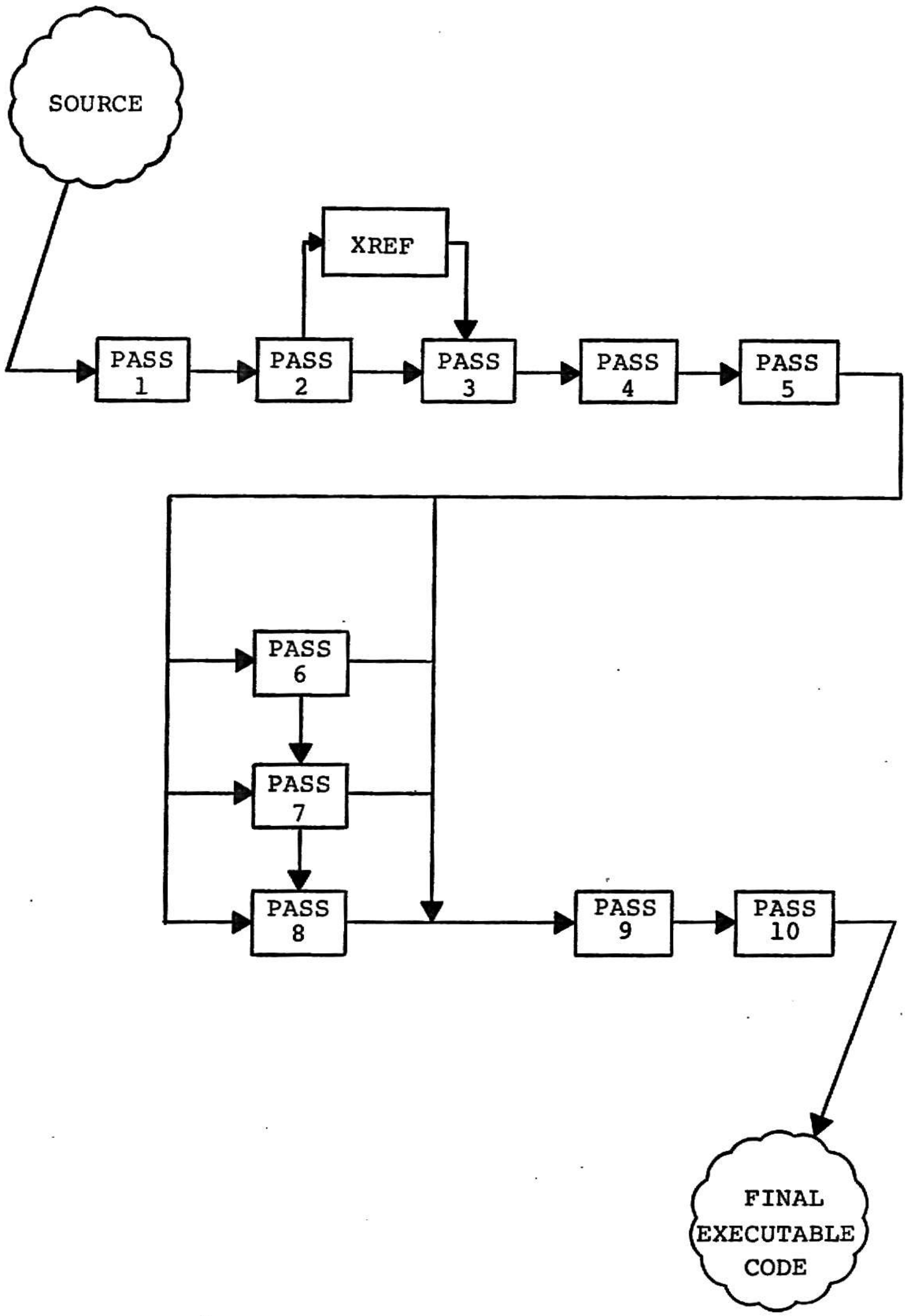


FIGURE 5: DATA FLOW--PASCAL/S COMPILER

time.

Expression Evaluation is another of the optimizing passes. In this pass expressions are evalutated in an effort to increase execution speed or to reduce stack size by minimizing the number of temporary symbols on the stack. Boolean expressions are evaluated with emphasis on increasing execution speed and operand switching performed with the emphasis on reducing stack size. Operations on Boolean, relational and arithmetic expressions are evaluated for use with 'op immediate' instructions. This pass eliminates redundant operators.

Ad Hoc Optimization is the final optimizing pass. This pass includes rather diverse optimization features which did not fall into the categories of the previous optimizing passes. Briefly, this pass does optimization on control structures, removes all branches to branches, factors arithmetic expressions, removes subscripting, performs strength reduction [Grie71a], looks for identical operations and flags all branches to insure they can be reached.

Code Selection is the first of the code generation passes. It is similar in task to Hartmann's pass 6. However, it selects code for the stack machine instruction set as opposed to Per Brinch Hansen's virtual Pascal machine. The instruction set is different from that of Per Brinch Hansen and exhaustive. This pass also builds the four tables created in Hartmann's pass 6 (routine label table, a jump label table, a stack table and a constants table) which are then used in the next pass. Large constants are collected.

Code Assembly, the final code generation pass is essentially the same in concept as Hartmann's pass 7. In this pass proper displacements are inserted, large constants are placed at the end of the generated code, program labels are replaced with addresses and error messages are also written.

1.7: SCENARIO OF DOCUMENT

The remainder of this document is structured in the following way. The next chapter presents a discussion of the language modifications and extensions of Hartmann's compiler: first, those modifications added at Kansas State University are described; second, the additional modifications for the PASCAL/S version are developed. Chapter 3 presents a discussion of optimization which includes discussion of constant folding, Boolean and arithmetic expression evaluation and other optimization. Chapter 4 is devoted to code generation (code selection and code assembly). The final chapter contains concluding remarks.

In addition the following appendices are included for benefit of the reader:

- Per Brinch Hansen's Sequential Pascal Report as modified to include changes incorporated in the PASCAL/S compiler.
- Syntax graphs for input to all passes and the final code for the PASCAL/S compiler.
- A User's Manual including a discussion of all program options.

An annotated Bibliography of Pascal and related resources as well as references from this paper is also provided.

LANGUAGE MODIFICATIONS AND EXTENSIONS

2.1: Kansas State University version

2.1.1: Changes made prior to PASCAL/S

Most of the modifications made to the Hartmann compiler at KSU prior to the development of the recent version, were designed to facilitate using the Pascal source language and represent only minor changes in pass 1 and pass 2.

The language modifications and extensions added to the KSU implementation of Hartmann's SPASCAL compiler prior to the development of the PASCAL/S version are the following:

1. The **character set** was extended to include all lower case letters and these additional symbols: {, }, [,], and ∧. The addition of lower case letters simply extends the readability of the language and necessitated relatively easy modifications. The 'up arrow', ∧, may be used in addition to @ to denote pointer type or pointer component. The other special characters added to the character set are included in Pascal's special symbols and discussed in the next paragraphs.
2. Several **special symbols** were also added. They are (*, *), {, }, [and]. These special symbols have fixed meaning unless they appear within

string constants or comments.

3. Among these new symbols are those used as new **delimiters for comments.** In addition to the double quote, ", used in Hartmann's version [Hart75a], the following symbol pairs (* and *) and the braces, { and } may now be used. (See syntax graphs in Appendix B.) Whichever symbol or symbol pair that begins the comment determines which symbol or pair of symbols must end the comment. The symbol(s) ", *) or } may not be included in the comment when it is the same as the one to delimit the end of the comment.

4. The square brackets, [and], which were added to the special symbols may be used to replace (. and .) respectively within **array type definitions** or array components to denote **subscripts** and with sets to denote the construction of sets and to delimit sets.

2.1.2: Changes Made for XREF

The cross reference (XREF) is a feature added to Hartmann's SPASCAL compiler by Anderson [Neal76a] at KSU. Modifications to the compiler to accommodate the XREF feature include additions to pass 1 and the creation of a new pass.

The XREF pass is inserted between pass 2 and pass 3 of the Hartmann compiler. (Shown in figure 5.) Thus the input to the XREF pass is the same as the output from pass 2. The execution of the XREF pass leaves the intermediate code unchanged for input to pass 3. The XREF feature is optional and must be specified in the program's options (See User's Manual, Appendix C.)

Modifications to pass 1 include the addition of a procedure called DUMPSPELLINGS and related other procedures. Dumpspellings primarily produces two tables called 'IDSPELLINGS'. One table contains only the unique names which are in sorted order by their internal representations. This internal representation is the identifier's spelling index (spix). The other table is alphabetically ordered and contains the spix in addition to the name. (See figure 6.) The procedures supplemental to DUMPSPELLINGS include those necessary to build and traverse the binary tree used in creating the alphabetical ordering of identifiers. The two tables created in pass 1, represent IDSPELLINGS and are later accessed by the XREF pass.

The main objective of the cross reference pass (XREF) is to produce an alphabetical listing of all identifiers used in the source. Each entry would include the type of the identifier, the number of the line where it was declared, and a list of line numbers where the identifier was used.

In order to attain the useage line numbers, this pass performs functions similar to scope analysis of pass 3. Rather than replicating the update and display data structures which are actually used in pass 3 and described later in this chapter, all entries are saved on the stack with the newer, active entries nearer the top. (See figure 7.)

When a new block is entered, the entries for the previous block are not popped and lost as is logical in scope analysis, but rather these entries are saved in a list. To update the usage of a particular identifier, a search is initiated via the display stack which maintains a pointer to the beginning of each block currently on the stack.

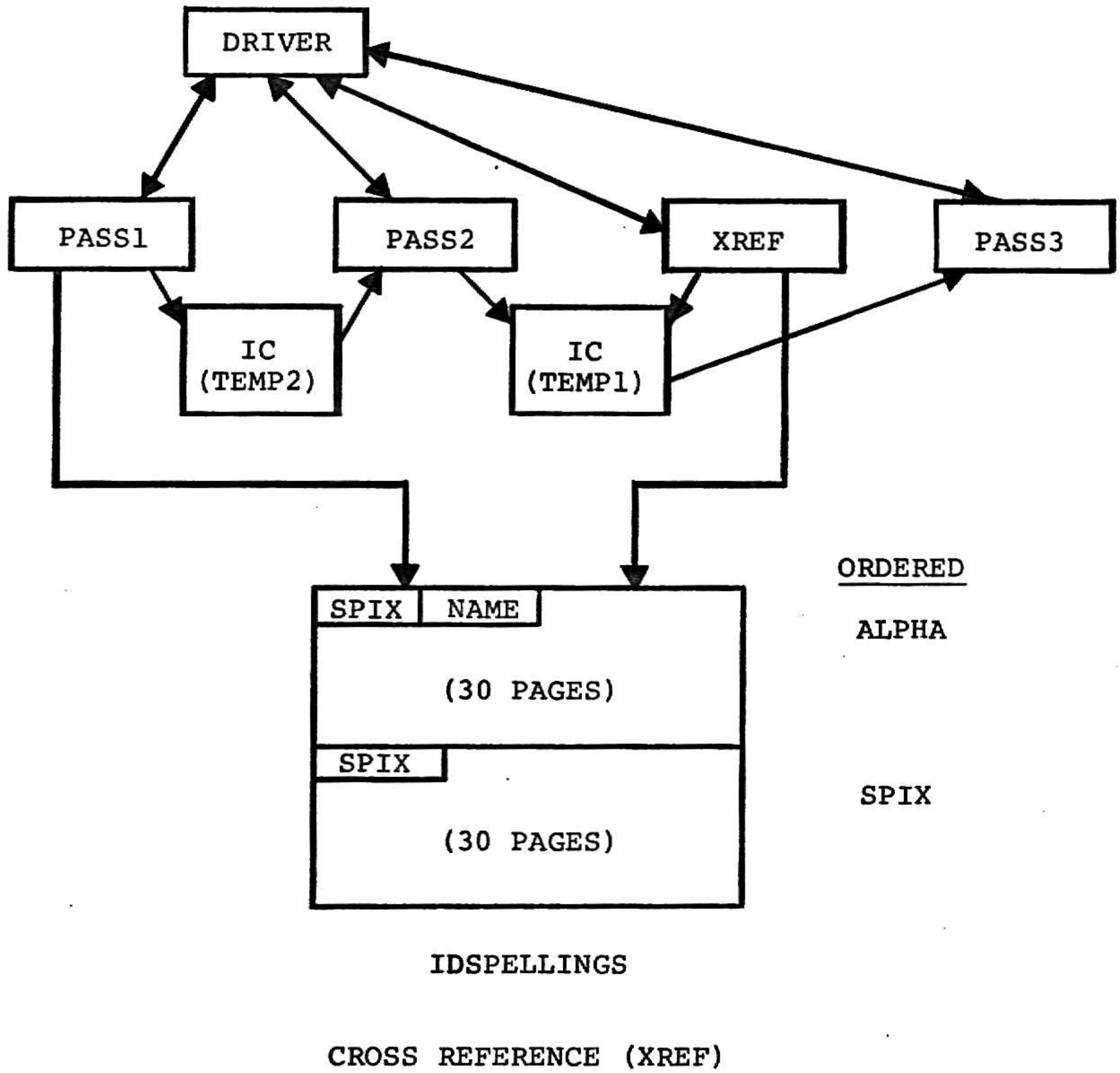


FIGURE 6 : CROSS REFERENCE--DATA FLOW & FLOW OF CONTROL

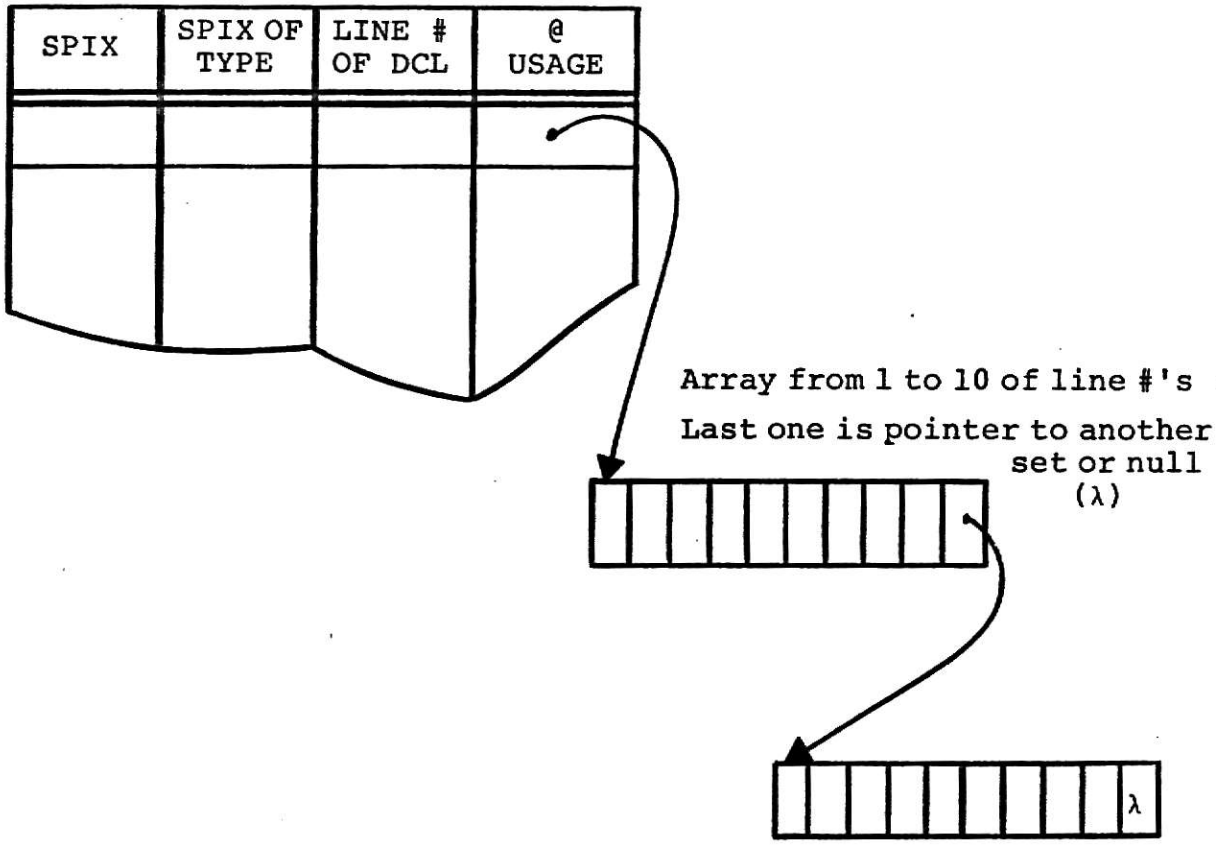


FIGURE 7 : STACK DATA STRUCTURE USED IN XREF PASS

Once the cross reference list is completed, 'IDSPELLINGS' is used to alphabetize the list for output. The portion of 'IDSPELLINGS' which is numerically ordered by spix is used to attain the name of the identifier. Since all identifiers are truncated to twelve characters and the spix itself represents the displacement into the table, the name may be accessed directly by taking twelve times the spix and adding that to the beginning point of the table.

The more common names, eg. INTEGER, ARRAY, etc., are saved in core and not accessed in this manner. It would be desirable from the point of view of speed to have the entire list in core; however space limitations interfere with this possibility.

Conceptually, there are four tables used in the XREF pass.

- the stack of identifiers
- the display stack
- the output list
- IDSPELLINGS (created in pass 1)

The alphabetically ordered portion of 'IDSPELLINGS' is used to facilitate the alpha ordering of the output list. For each entry in the alpha listing the 'output' list is searched until a corresponding name is found. Once found that entry is output in the XREF listing.

2.2: The PASCAL/S Version

2.2.1: General Description of Changes

Specific modifications and extensions made to the Hartmann compiler for the FS version include the following:

1. Hartmann's compiler was extended to include **set constants** which consist of finite binary strings. Thus, a set constant will consist of a binary string whose length may be either 32, 64, or 128 bits and these must be completely specified. For convenience a repetition factor (See syntax graphs, Appendix B) may be used to facilitate completely specifying the binary string.

2. **Set type** has been implemented in a slightly different way. When used, the type is analyzed such that the most efficient set length among 32, 64, or 128 bits is selected. Strictly speaking, it is the ceiling of the set length which is large enough to encompass the type.

3. Another type, **SREAL,** has been added. SREAL denotes a short real type. That is, a type which identifies any of the subset of reals which can be represented in 4 bytes rather than 8 bytes.

4. **Negative constants** may now be used in virtually all places in the program (eg. in FOR statements, as subscripts, as limits on subranges) where constants may be used.

5. Type **INTEGER** has been modified, so that the use of INTEGER as a type is now synonymous with either short, standard, or long enumeration, depending upon the option specified. The default is 16 bits (2 bytes). A short INTEGER is defined as a 16-bit enumeration (-32768..32767). A standard INTEGER is a 32-bit enumeration (-2147483648..2147483647). A long INTEGER is a 64-bit enumeration (-2 exp (63)..2 exp (63) - 1).

6. A new facility of **COERCION** has been added which affects evaluation of arithmetic expressions and the values associated with parameters.

When evaluating arithmetic expressions all variables' values are coerced (expanded) to the length of the longest value in the expression. In addition, all integer constants are coerced to the length of the longest value in the expression and all real constants are coerced (truncated or expanded) to the length of the longest variables' value in the expression.

Coercion is also used in the evaluation of parameters. When the formal parameter is of type VAR, then no coercion takes place. However, if the formal parameter is of type constant then actual parameters which are shorter are coerced to the length of the formal parameter. If the formal parameter is of type constant then actual parameters which are shorter are coerced to the length of the formal parameter. If the formal parameter is of type short real constant then any actual parameters which are long real are coerced (truncated) to the length of the formal parameter.

7. **ESCAPES** from the language are now provided so that micro-code or assembler code can be executed. ESCAPES allow for efficient execution of highly used routines.

8. The internal representation of **pointers** was changed from 16 bits to 32 bits. The internal representation of characters and Booleans was also changed from 16 bits to 8 bits.

2.2.2: Specific Changes to Hartmann's First Five Passes.

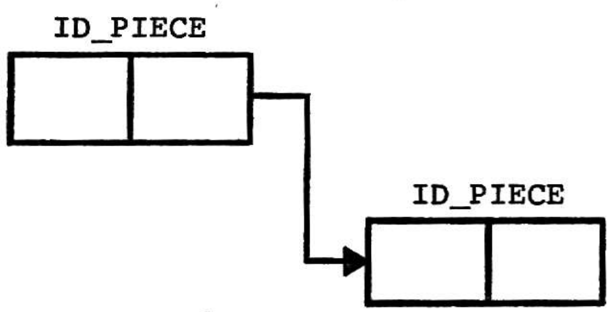
Specific changes to each of Hartmann's first five passes are discussed in the section which follows by examining each of these passes in order. Some of the material in this section has been paraphrased from Hartmann's dissertation [Hart75a].

Pass 1: Lexical Analysis. Lexical analysis provides the interface between the source program and the remaining passes of the compiler. In this pass the lexical analyzer or scanner reads the source language one character at a time and produces sequences of characters often called tokens. Each token or symbol represents one logical entity. This conversion yields the first intermediate code.

The lexical analysis phase can be represented by a deterministic finite state automaton. This implies that each symbol type begins with a unique character or set of characters (e.g. identifiers begin with letters, string constants begin with quotation marks, etc.).

Primarily this pass scans identifiers, special symbols, and numbers. The lexical analyzer must be capable of recognizing the extended character set. As documented by Hartmann [Hart75a], "the scanning of an identifier consists of collecting the identifer in a string variable, searching for it in a table of identifiers, and outputting the corresponding intermediate code." (See figure 8.) Identifiers may be either reserved words or program defined identifers. When the identifier is a reserved word, the intermediate code output is the operator corresponding to it. For all other identifiers the intermediate code is an ID operator followed by the spelling index (spix) of the identifier. The spelling indices for unique identifiers are assigned sequentially beginning with the first number following the index of the last standard spelling noun.

The spix of an identifier is used as an index into the spelling table. Each entry in the spelling table is an index into the hash table. (See figure 9.) This hash index or hash key is created by using the ordinal value of each character of the identifier as it is read. This hash function computes the product of the ordinal values of the characters within an identifier modulo the table length. Stored in the hash table are the identifier's spix, the first ten characters of the identifier and a pointer to additional pieces of the identifier if longer than ten characters. The maximum length of an identifier is 80 characters. Reserved words are not found in the hash table since they are identifiers whose indices are negative.




```
TYPE
  PIECE = ARRAY[0..ID_PIECE_LENGTH[ OF CHAR;
  PIECE_PTR = @ ID_PIECE
  ID-PIECE = RECORD
                 PART:  PIECE;
                 NEXT:  PIECE_PTR
               END;
```

FIGURE 8 : IDENTIFIER ENTRY

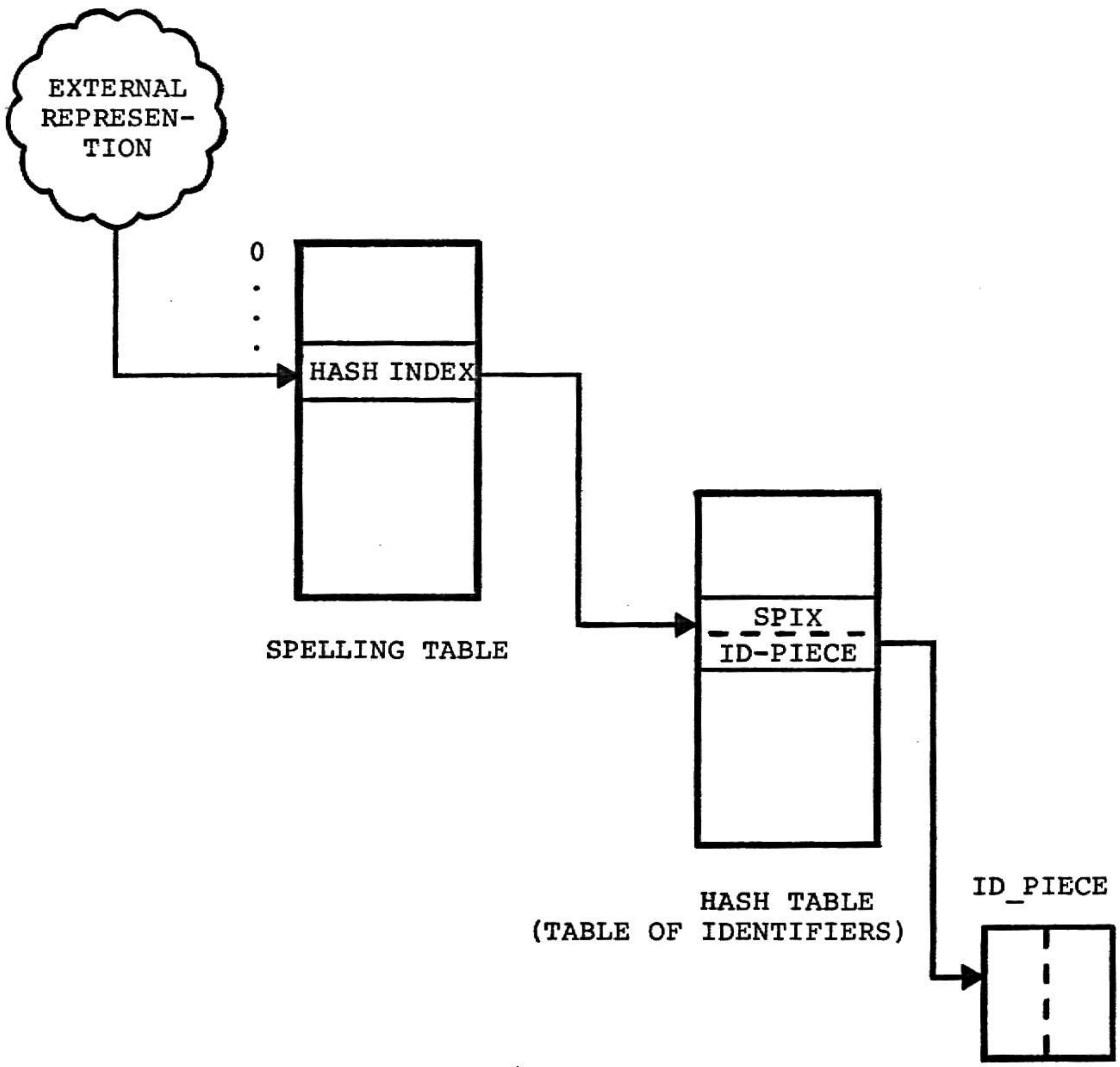


FIGURE 9 : HASH TABLE

Since it is predictable that different identifiers may yield the same hash index, when this does occur, a cyclical search is begun. The search terminates when either an empty entry in the table is encountered or the identifier, itself, is found to be in the table.

Given that Pascal requires fixed length arrays, the identifier table is of a fixed length. Because it is anticipated that some hash indices will duplicate others, the table is limited to a 98 percent fill by the compiler to prevent the resulting long searches. When the addition of a new identifier exceeds this limit, an insert error is generated and lexical analysis is terminated. Thus only that portion of the intermediate code is transmitted to succeeding passes.

The scanning of special symbols produces intermediate code consisting of its constant representation. Specific to the PASCAL/S modifications, the new special symbols which delimit comments, indicate array types or array components and signify a pointer type or pointer component must be recognized and transformed into the appropriate intermediate code.

The scanning of numbers varies somewhat from that of the Hartmann compiler. In the PASCAL/S version when the first digit is encountered, it is assumed that the number to be scanned is an integer. This assumption is made because the potentially largest integer is larger than the largest real number in the PASCAL/S version. Therefore, the digits are collected into an array of integers. Integers are saved as the shortest integer possible while all reals are saved as the longest real.

Once either a decimal point or an 'E' is encountered, the existence of a real number is confirmed. However, the scanning process continues to collect both the fractional part and/or the exponent part as arrays of integers. It turns out that the fractional part is treated as a continuation of the integer part and the exponent is adjusted accordingly.

The intermediate code produced when an integer is scanned is: the token (constant) for 'integer', followed by the token for 'large constant', followed by the length in bytes of the integer, followed by from 2 to the length in bytes of the integers, which represent the value of the integer (e.g. 2 yields 11 <integer> 59 <large constant> 2 <length> 0 2).

When real constants are scanned the intermediate code produced is similar to that of integers. First is the token for 'real' followed by the constant for 'large constant', followed by the length in bytes followed by from 4 to the length DIV 2 integers which represent the value of the real (e.g. 1.2424 yields 9 <real> 59 <large constant> 8 <length> 16659 -7970 25 0). The length for reals is always either 4 or 8 bytes.

When set constants are scanned, the intermediate code produced is similar to that for reals and integers. First is the token for 'sub' followed by the constant for 'set constant', followed by the length in bytes followed by either 2, 4, or 8 bytes (depending upon the length) which represent the value of the set (e.g. [B'(32)'1] generates 15 <sub> 63 <set constant> 2 <length> -1 -1 47 <bus>).

Pass 2: Syntax Analysis. Syntax analysis accepts as input the first intermediate code, i.e. the output of lexical analysis. It is the function of this pass to check that the tokens input occur in acceptable patterns as defined by the syntactic specifications of the language (see syntax graphs, appendix B). The intermediate code which is output from this pass is syntactically correct. If the input is incorrect then the resulting code, although syntactically correct, may not be especially meaningful.

One of the most common parsing techniques, recursive descent, is used to perform the syntax analysis. This technique uses a set of recursive routines to perform syntax analysis. Each of the syntactic constructs in the language represented by the syntax graphs (See Appendix B.) is handled in a separate, possibly recursive, procedure.

Error recovery, which is also directed by the syntax graphs is covered in detail in Hartmann's dissertation [Hart75a] and will not be discussed in this report.

Other accomplishments of this pass include converting the code from infix to postfix notation, replacing ambiguous operators by unique ones and eliminating redundant operators. Specific to the PASCAL/S version, this pass recognizes the escape construct and set constants. Escapes are implemented similar to prefix calls. The escape mode is then handled in the recursive descent procedures. Set constants are implemented similarly to other constants. Perpetuated in this pass are allowances for negative constants, different size sets, different size subranges and short reals -- essentially all the general changes described previously.

Pass 3: Name Analysis. Name analysis accomplishes several chores. It transforms spelling indices into name indices while enforcing the scope rules. Thus each name index refers to a single specific entity (type, constant, variable, etc.) throughout its existence. The scope rules enforced in this pass place some constraints on the recognition of identifiers.

Before an identifier can be recognized it must have been defined. In order to be defined it must have been introduced which may be done through declaration or qualification. If introduced through declaration, then a specific type, constant, variable, or routine is associated with the identifier. Qualification which occurs when the variable name is followed by a period or by using a WITH statement, associates a field with a particular record variable.

The scope rules germane to this version are borrowed from those delineated by Hartmann [Hart75a].

1. An identifier is only known with a given meaning after its introduction (with that meaning) and until the completion of the body, record, or qualification that introduced that identifier (with that meaning).
2. No identifier may be given more than one meaning in a single block or record.
3. An identifier may be introduced with another

meaning in another block, record, or qualification. Where this occurs, the new meaning applies until the completion of the block, record, or qualification.

4. Within a routine, in addition to the above, all identifiers introduced in that routine are known.

Three tables are created in this pass to assist in enforcing the scope rules.

The **SPELLING TABLE** translates the spix into a unique name index and enforces the scope rules by attaching the nesting level index to it. The spelling table is quite large since it must contain space for the entries of all of the possible spixes from 0 to 700. (See figure 10.)

The **UPDATE STACK** is another of the tables used in this pass. Primarily, the update stack is used to save a previous entry in the spelling table whenever the name is encountered within a new block or scope. Where a block implies procedure, function, or with statement and the main program constitutes the first block. This technique was borrowed from Naur [Naur63a] in Hartmann's implementation. At the end of the scope, any 'old' entries are popped from the update stack and returned to the spelling table.

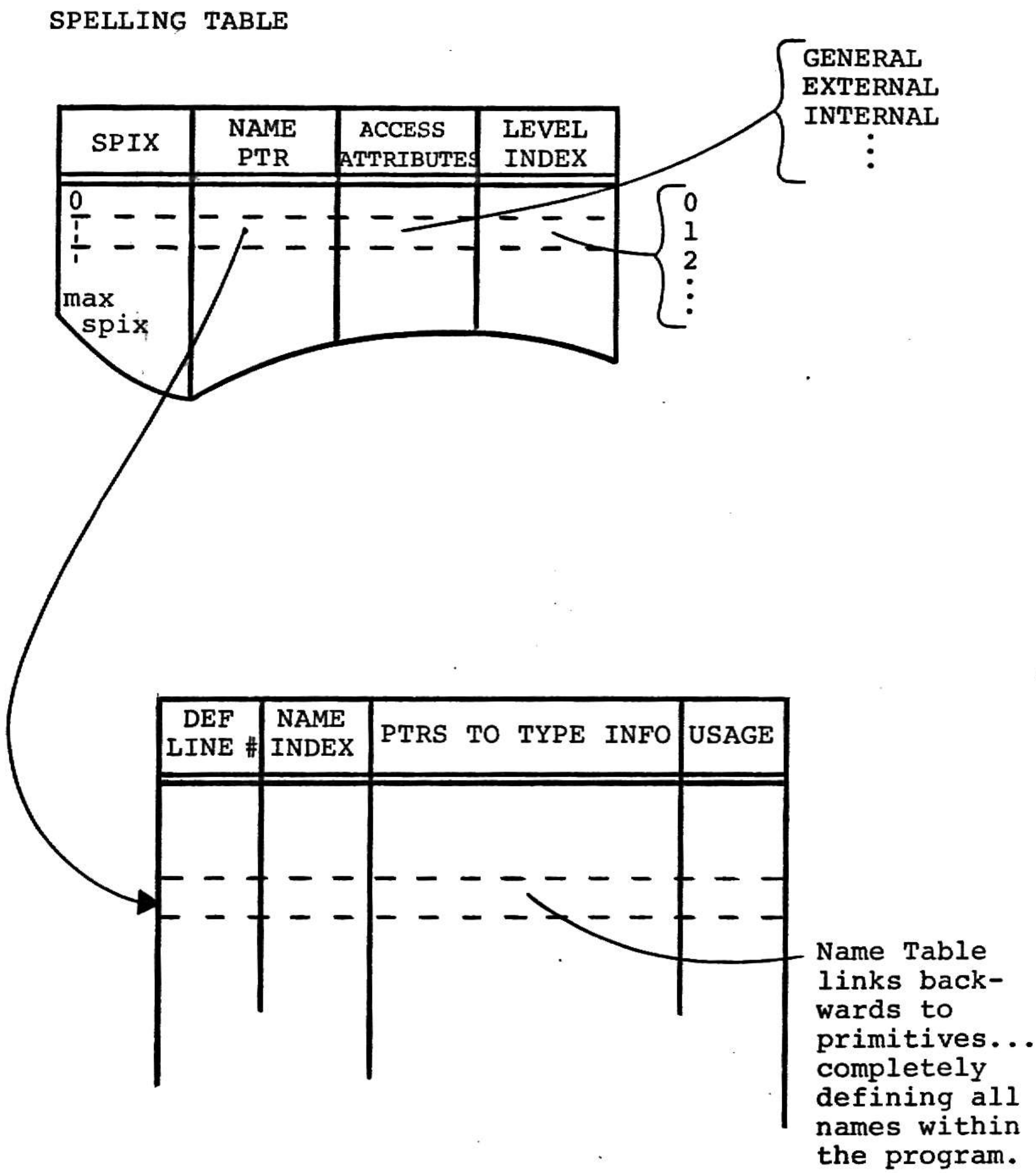


FIGURE 10: SPELLING TABLE & NAME TABLE

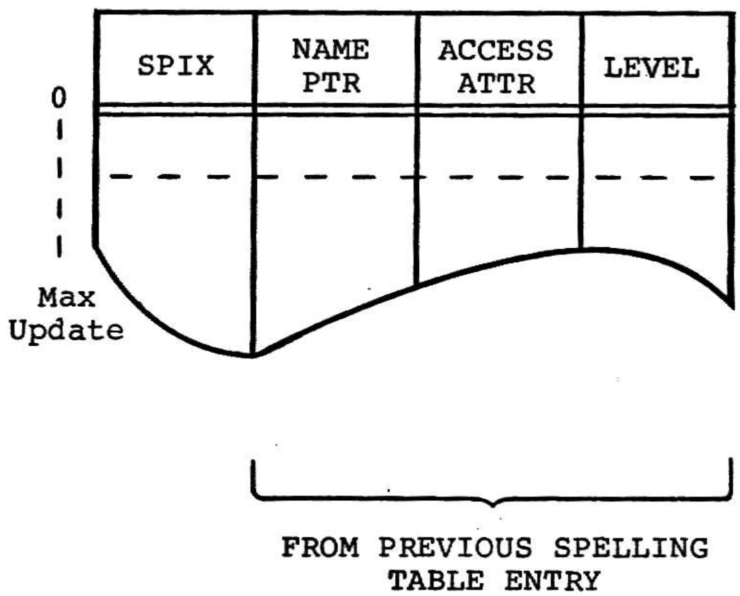
However, this requires that some record be kept of where any new scope's update stack begins. (See figure 11.)

This resulted in the use of a construct similar to a **compile-time display,** the third of the tables created in this pass. (See figure 12.) Basically this table stores the base indices of the block entries in the update stack and thus contains information related to the nesting levels. For each new level entered via declaration or a WITH statement, a new entry is pushed on the display table.

The update stack may be relatively small since it needs to include only space for local variables or qualified variables which have the same name as a previously defined variable. The information saved in the display provides for faster access into the spelling table.

In the PASCAL/S version access to escapes is provided from anywhere in the program. Escapes are implemented in the PASCAL/S compiler as entries in the prefix.

There is another data structure essential to name analysis, the name table. It contains all information associated with a name which is pertinent to name analysis. (See figure 13.) For every name which is recognized throughout the spelling table, there exists a pointer into



THIS TABLE IS RELATIVELY SMALL. IT CONTAINS ONLY LOCAL VARIABLES WHICH HAVE THE SAME NAME AS SOMETHING OUTSIDE.

FIGURE 11: UPDATE STACK

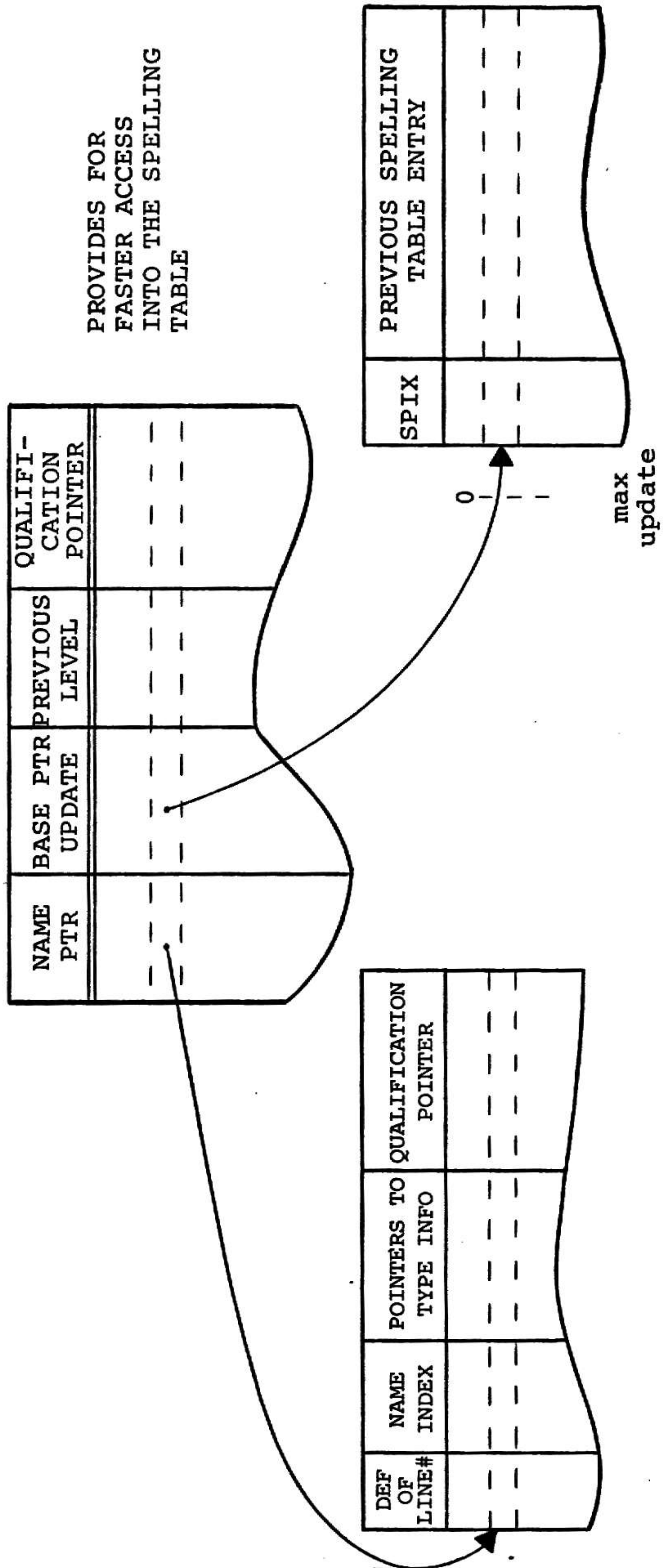


FIGURE 12: DISPLAY

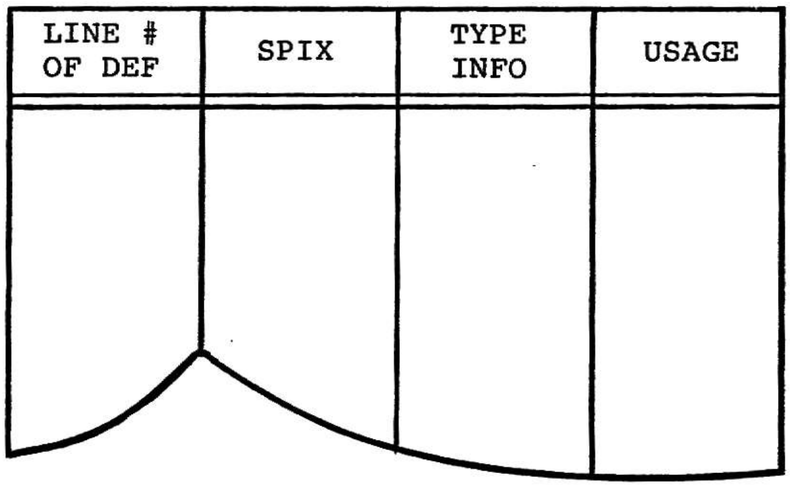
the name table. Information is stored about each name whether it is the name of a type, constant, variable, parameter, or routine. The name table is actually a linked list which represents the access relationships of types, variables, parameters, and routines.

Since constants are nameless, they have no name index. Name analysis proceeds to remove all constant declarations from the intermediate code. Thus, all constants except string constants are represented in the name table by their value. String constants are represented in the name table by their displacement in the program's large constant area as in Hartmann's version.

It turns out that all types are represented by a name index regardless of whether they were actually named by the programmer. This facilitates the declaration analysis which is to be done in the next pass, since all types may now be referred to by their names (name indices).

In summary, pass 3 name analysis is responsible for assigning name indices to types, variables, parameters, and routines and replacing constants by their values (or displacements in the case of string constants).

Any reference to a name now implies looking in the name



- Always scanned from the top-down unless directed otherwise by displays
- Built by analyzing DCL portions or routines when DCL statements are translated
- Used to analyze statements within program

FIGURE 13: SYMBOL TABLE

table. The relationships between/among subrange types and their range types, routines and their parameters, functions and their result types, and the program and its interface, arrays and their index and element types, the with statement temporaries and their record types, record types and their fields are all represented by links within the name table. The information from the name table is then placed in line in the output code as necessary.

Name analysis also utilizes an operand stack which stores operands because they precede their operator. Each entry on the operand stack is similar to an entry for a name in the name table with minor differences.

Listed below are the outcomes associated with an operand related to specific constructs:

Constants:

- when encountered in a declaration, are pushed on the op stack
- when in constant definitions, are placed in the name table
- when constant labels, are pushed on the operand stack as case labels
- when factors, are immediately placed in the intermediate code and an empty entry is pushed on the operand stack;

Variables:

- when referenced in a body, imply that the variable type is pushed on the operand stack,
- if 'subscripted', then the name of the array element type replaces the name of the array type,
- if 'qualified', then the field type replaces the record type;

Routines:

- when referenced in the body, are placed on the operand stack. (The operand entry contains the name of the routine and the name of its first parameter.)

Names are declared in a declaration part. While the declaration is still incomplete, the operand stack entry indicates a declaration. Associated with the declaration are its spelling index and a pointer to its incomplete name entry. This information is used to update the various tables at the completion of the declaration.

The accomplishments of pass 3 are:

- all types, variables, parameters, and routines have unique name indices which are used by later passes,
- all linkages between these entities have been checked and now appear in the output code,

- the name table represents the structural relationships of language elements.

It turns out that the major complexities of the language are contained in the name table. Thus name analysis has accomplished its primary task of analyzing names related to scope and establishing correct relationships. Once name analysis is done, the remaining semantic passes can continue processing with some of the major complexities already resolved.

Pass 4: Declaration Analysis. Declaration analysis, the second of the semantic analysis passes, performs the semantic checking of the declaration portions of the program. As documented by Hartmann [Hart75a], this pass "analyzes types, assigns addresses to variables and parameters, assigns program labels to routines, and distributes this information in the body parts."

Essential to the enforcement of the numerous semantic rules is the recording and bookkeeping of all type information in the data structures of the pass. Requisite to declaration analysis is a new symbol table. The symbol table of this pass has no pointers, since all the links were analyzed and placed in-line in the previous pass. Thus in place of the links the input to this pass contains name indices. The name indices are translated through the name table which contains only pointers to the symbol table. (See figure 14.)

Declaration analysis proceeds in the following way: When a name is declared, an entry is added to the symbol table and the link inserted in the name table. Any future references to the name are handled indirectly through the name table.

The name table must contain an entry for each name index; however the symbol table entries are allocated dynamically as the declarations are encountered. Thus, the symbol table uses only the space it requires which implies that smaller programs may be compiled using less memory space than larger programs.

Also in this pass is an operand stack implemented as a one-dimensional array. The operands, which are name indices upon input, are translated into pointers to the symbol table via the name table. Stored in the operand stack are the pointers (links) to the symbol table. (See figure 15.)

In this pass there are four variants associated with the symbol table. The first variant contains variables and parameters combined (values). The second variant is used

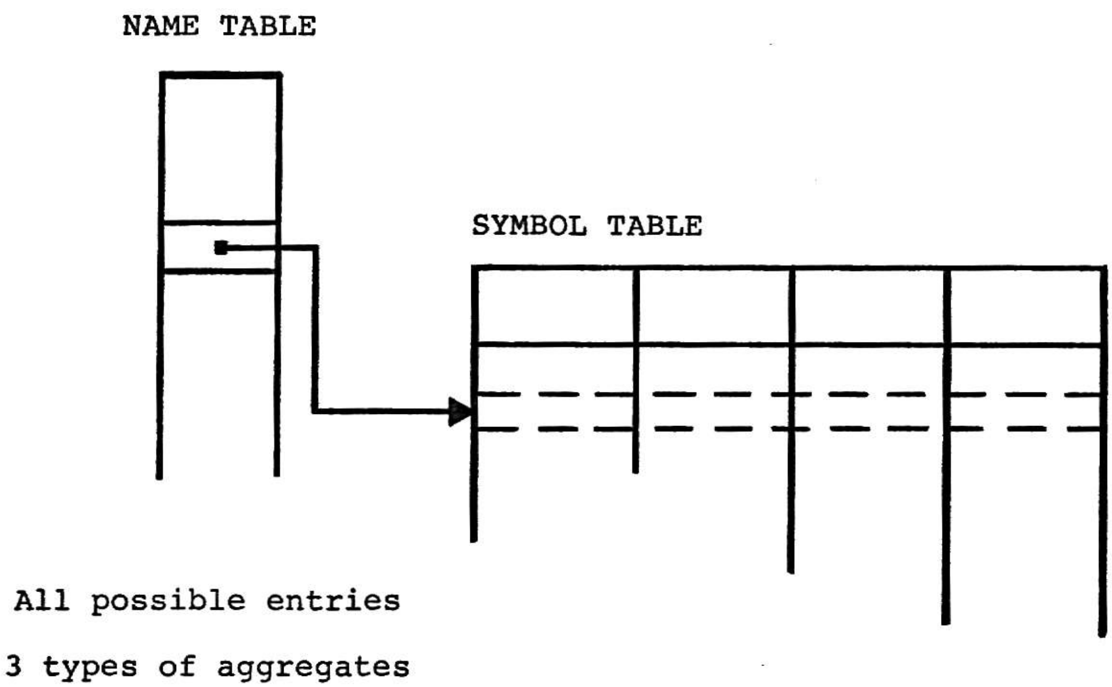


FIGURE 14: NAME TABLE & SYMBOL TABLE

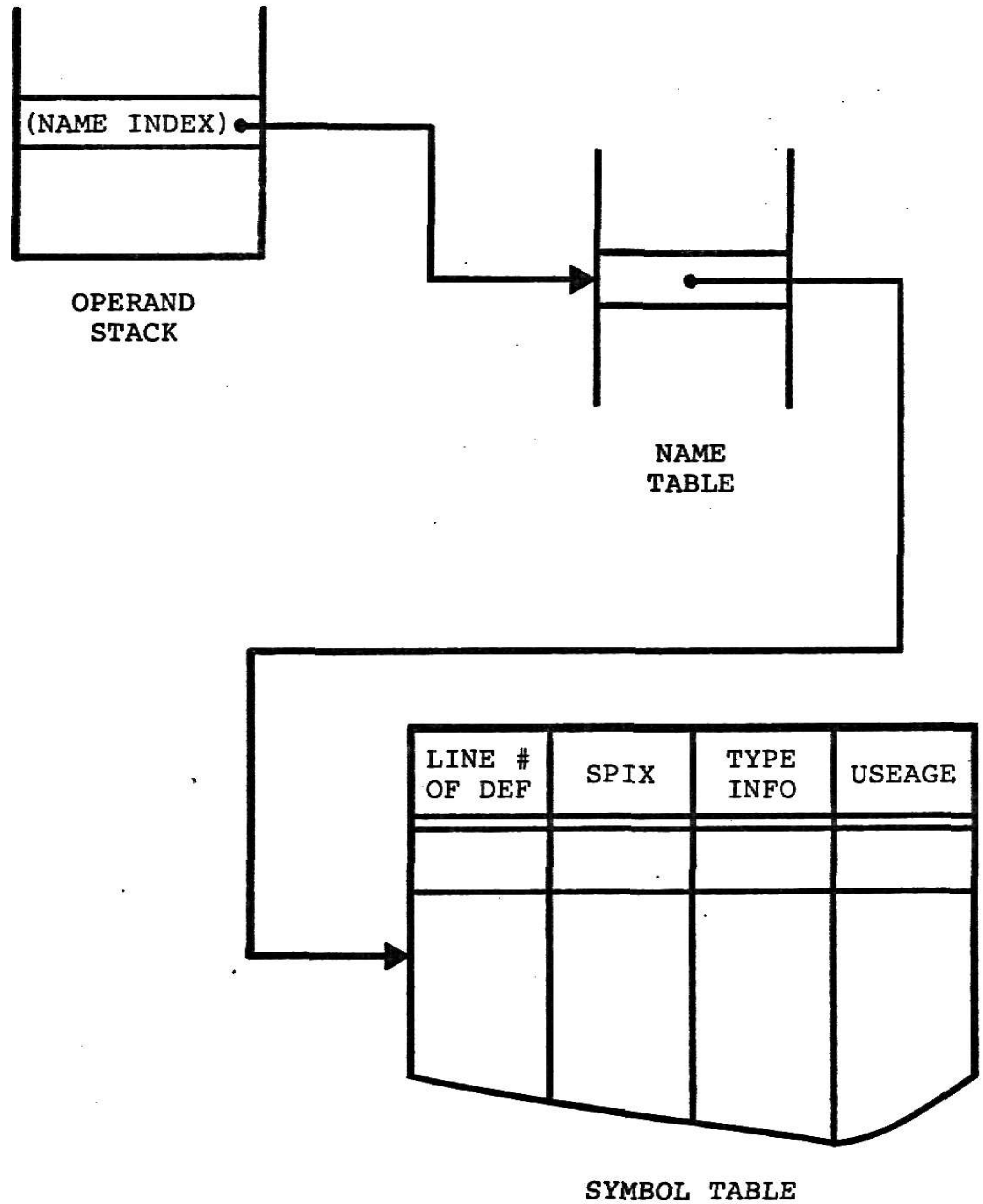


FIGURE 15: OPERAND STACK

for routines. The third variant is strictly for types (templates). The fourth variant is undefined and remains empty.

The symbol table entry for variables and parameters is represented as a value variant which contains the following information about the value:

- the address mode,
- the address displacement, and
- the declaration context.

Since this information is required by later passes, it is distributed in the output.

The address mode and address displacement combine to indicate the virtual address. The mode encodes information about the level of nesting and information about the entry routine. The modes available are:

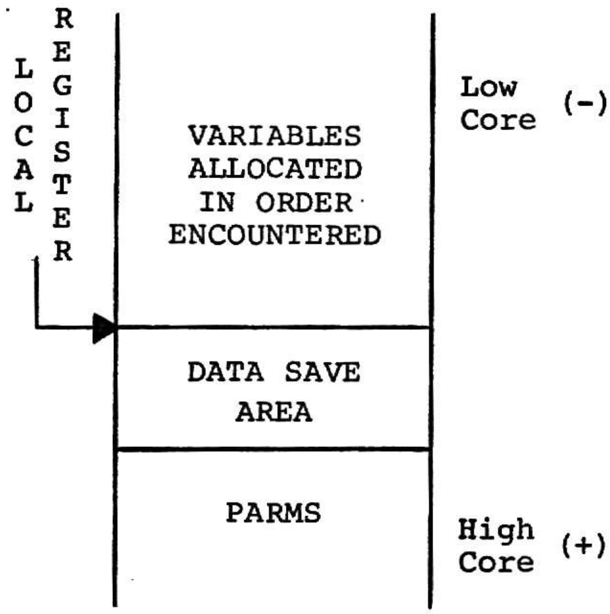
0. large constant
1. process
2. program
3. process entry
4. class entry
5. monitor entry
6. process
7. class
8. monitor
9. standard
10. undefined
11. string constant
12. escape entry

However, not all of these are used in Sequential Pascal. They are included for consistency with Concurrent Pascal.

Those actually used in this pass are: large constant, procedure entry for prefix calls, program, and procedure for routines. Thus, the mode includes information which establishes the appropriate base register.

The address displacement is the actual displacement of a value within the data area of a routine, record, or the constants area. These displacements are determined during this pass and assigned sequentially as the declarations of fields, variables, and parameters are processed. The actual displacements may be either positive or negative, with or without offset. The displacements are always related to a particular routine or record. (See figure 16.)

```
PROCEDURE (P1: INTEGER; P2: BOOLEAN);
VAR V1:  BOOLEAN;
    V2:  CHAR;
    R1: RECORD
         E1:  INTEGER;
         E2:  BOOLEAN
        END;
```

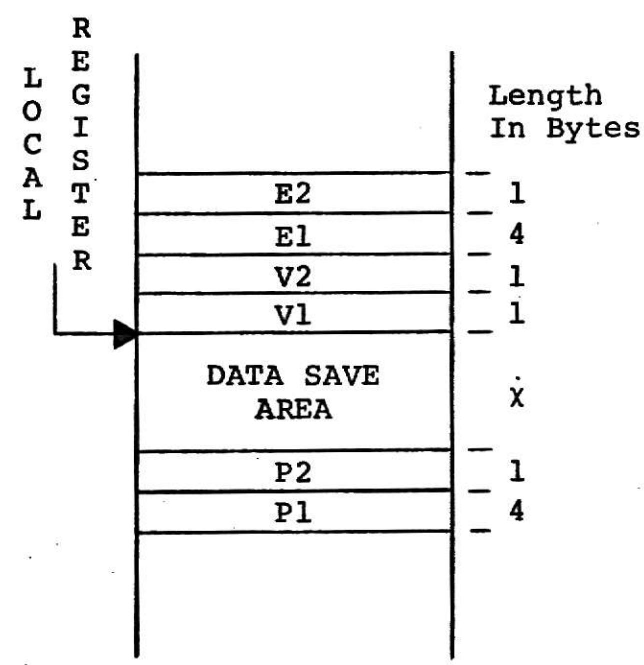


Example 1: Access E2 in R1

- First push address of R1 (PUSH LOCAL ADDE D2)
- Next access field within record (FIELD 4)
- This leaves the address of E2 on top of the stack

Example 2: Access P1

- Push address of P1 (PUSH LOCAL ADDR -D(χ+1)Ø)



χ represents the fixed length of the data save area--In Sequential PASCAL the data save area occupies the space of 5 pointers + 1/2 word.

FIGURE 16: ADDRESS DISPLACEMENTS

The declaration context identifies the context in which the value was declared. The possible contexts include the following:

1. function result
3. variable
4. variable parameter
5. universal variable parameter
6. constant parameter
7. universal constant parameter
8. record field
10. expression
11. constant
12. save parameter
13. new_parm
14. tag_field
15. with_const
16. with_var

The 'constant' and 'expression' contexts shown above are flags for pass 5 since they have no declarations.

The routine variant is used to represent routines in the symbol table. Each entry contains the address of the routine, the length of parameters within the routine, and the length of local varitables. The modes of the routine variant are the same as those for the value variant. No displacement is given, for these remain unknown until code assembly. As a result, the routine label is used as the second part of the address. Code assembly then resolves these labels and generates final code.

The parameter length, which is used to pop the parameters from the stack as each routine is exited, and the local variable length, which is used to create (push) storage area on the stack as each routine is entered, are recorded during routine declaration.

Thus, the symbol table entry for routines contains the following information:

- the routine mode
- the routine label
- the parameter length
- the variable length

The template invariant contains all information associated with types. The symbol table entry for template invariants include the following:

- the name index
- the type length
- the active attributes
- the type 'kind'
- information related to specific kinds

The name index is saved and passed along in the intermediate code for use in type checking which is done in body analysis. The length of the type is also saved for use in assigning displacements.

The type kind represents a classification into kind for all types. The possible kinds are

0. integer
1. real
2. Boolean
3. character
4. enumeration
5. set
6. string
7. non-list
8. pointer
9. list
10. generic
11. undefined
12. routine

Primarily these are chosen to assist with type checking of pass 5.

The accomplishments of this pass include placing the symbol table entries created by declaration analysis in-line in the intermediate code as needed. Thus, one entry from the symbol table may appear many times in the code output, since it is inserted wherever the entry is referenced within the program. Symbol table entries appear in the output codes in one of two formats, either the value or the routine format. The type information (template variant) uses the value format.

There are no major modifications to pass 4 for the PASCAL/S compiler. The ability to handle larger constants in-line has been incorporated as well as symbol table entries for all new types. Some new information which had been saved until the analysis of type declarations is performed is now placed in line.

Pass 5: **Body Analysis.** Body analysis, the last phase of semantic processing does the semantic checking of the body parts of the program. That is, this pass checks the type compatibility of operands and operators, generates commands for the machine and distributes these commands in line. The code input to this pass consists of a series of bodies since constant declarations were eliminated during name analysis as constants were placed in-line, and type, variable, and routine declaration information was placed in line during declaration analysis.

The type checking done in this pass consists of checking the compatibility of operands with each other and checking the compatibility of operands with their operator. For example, the NOT operator requires a Boolean operand, while the addition operator requires that its two operands be compatible with each other and that they be arithmetic operands.

The type checking done in this pass may cause coercion of operands. This notion is new in the PASCAL/S version. Coercion is used to force compatibility of operands as needed. Coercion within arithmetic expressions implies that:

1. all variables' values are coerced (expanded) to the length of the longest value in the expression, by creating intermediate code which causes conversion operations in the target code,
2. all integer constants are coerced (expanded) to the length of the longest value in the expression, and
3. all real constants are coerced (truncated or expanded) to be the length of the longest variables' value in the expression.

Coercion as applied to parameter passing implies that:

1. if the formal parameter is of type VAR, the no coercion takes place,
2. if the formal parameter is of type constant then shorter actual parameters are coerced to the length of the formal parameter,
3. if the formal parameter is of type short real (SREAL) constant, then long real actual parameters are coerced (truncated) to the length of the formal parameter.

The type checking rules of Sequential Pascal were chosen to facilitate type checking and the learning of the language. The rules Hartmann delineated [Hart75a] have been extended and modified as appropriate for the PASCAL/S compiler. Thus, two types are compatible if any of the following is true:

1. they are defined by the same type definition,
2. both are subranges of a single type,
3. both are subranges with one of a longer type than the other, in which case coercion forces the shorter one to the longer length,
4. they are string types of the same length,
5. they are set types whose members are the same index type and are of the same length,
6. one type is a universal parameter type and the other type is an argument type of the same length,
7. one type is an argument type and the other type is its generic parameter type,
8. they are reals and one is a short real, which again causes the shorter one to be coerced to the longer length.

One additional note related to parameters for built-in functions is needed for clarity. Parameters for CHR and CONV may be of any integer type (INT2, INT4, INT8) and parameters for TRUNC may be any real (REAL or SREAL).

An operand stack is used in this pass to keep track of the type information used in type checking. All operands in Hartmann's version were 16-bit words. In the PASCAL/S version operands are of variable lengths, for example byte, long integer, shortsets, etc. Input to this pass is the type information as inserted in the code by declaration analysis. Thus, types are transmitted with three arguments: kind, name index, and length. These were conveniently chosen to 'mesh with the compatibility rules in a simple manner' [Hart751]. This is accomplished by using a small set of primitive attributes to represent the necessary type and context information of operands.

Most all operators and operands are tagged with their type in-line unless the type is implicit from the specific operator. For example:

```
     PUSHCONST  \
         TYPE    |
         VALUE   |
                  >  explicitly tagged with type
     PUSHVAR     |
         TYPE    |
         MODE    |
         DISPL  /
```

```
PUSHADDR    \
   MODE      |
   DISPL     |
              >  type implicit from operator
FIELD        |
   DISPL    /
```

Thus all type information is now retained in the intermediate code either implicitly or explicitly. Entries in the operand stack will have one of the following forms. (See figure 17.)

The address information ('mode' and 'displacement') describes the virtual address of the operand. For routines the 'displacement' is actually a label which is resolved during code assembly. Type information is identical to that described in declaration analysis. Value information is the same as that discussed previously except for 'state' (address state) which is to be described later in this section. Routine information is as was previously discussed. All displacements created in this pass and pass 4 are now 32 bits long.

When the full range of operand types is possible, type compatibility is checked by a function which compares the type of the top operand on the stack to the type of the

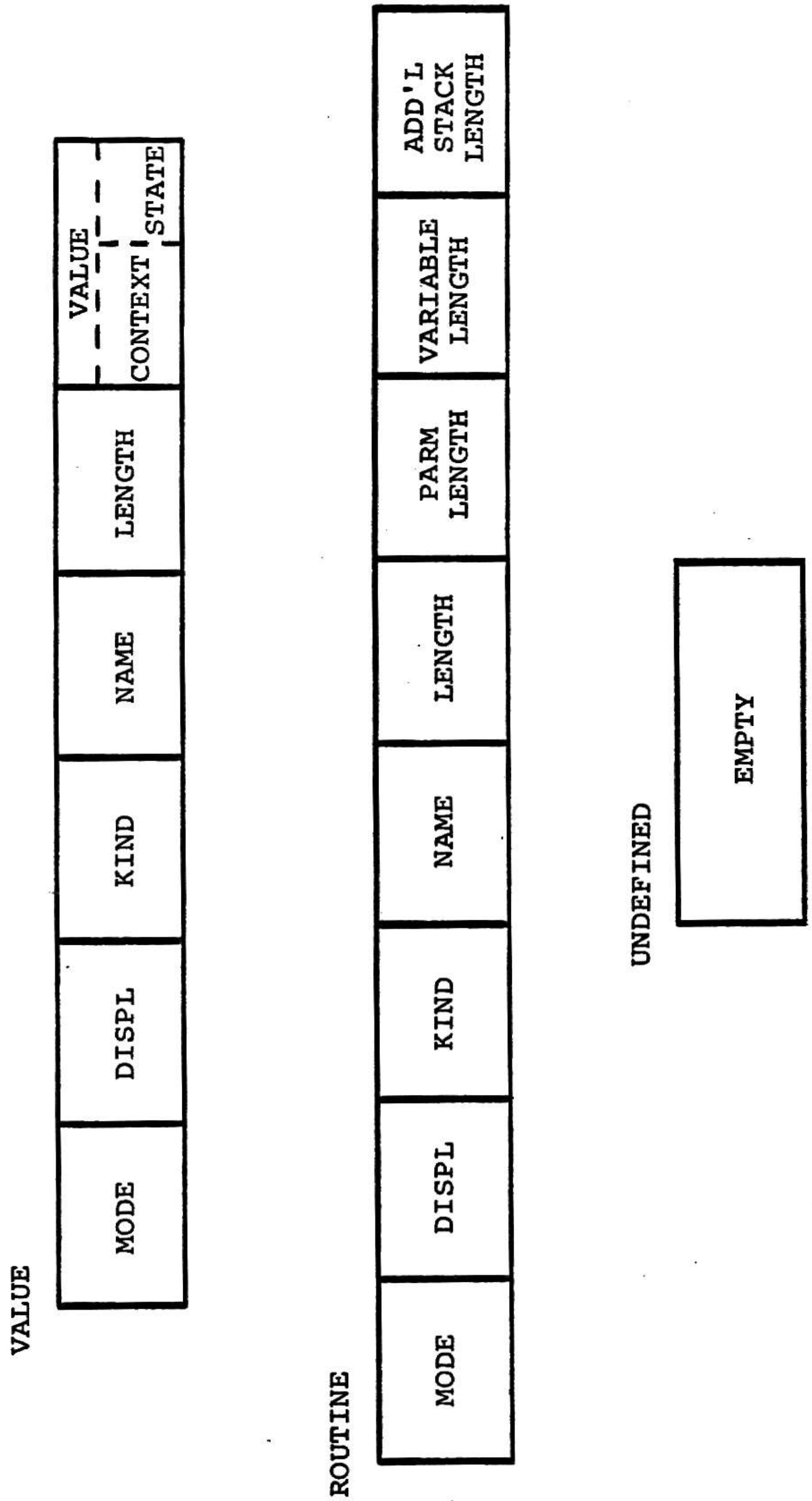


FIGURE 17 : ENTRIES IN OPERAND STACK

operand which is second on the stack. With most operators, however, specific operand types are required and type checking can usually be performed in line.

Type checking also makes use of the context and kind of a value to insure that assignment targets and variable arguments are assignable.

As mentioned previously, part of the value information is an address state. The possible address states are:

direct,

indirect,

addressed, or

expression.

The following definitions are borrowed from Hartmann [Hart75a]:

> The **direct** state indicates an operand that is directly addressable. Its mode and displacement are known. Unqualified variables and constant paramenters are directly addressable.
>
> The **indirect** state indicates an operand whose address is indirectly addressable, for example, a variable parameter.
>
> The **addressed** state indicates an operand whose address is on the machine's stack (such as a subscripted variable), while the **expression** state

indicates an operand whose value is on the machine's stack.

In the end, the machine requires that the state of an operand be either addressed or expression. In the PASCAL/S version, all operands are placed directly on the stack except operands of structured type (arrays or records), and string constants. Arrays and records are stored in the variable area (local or global) and their addresses are pushed on the stack. String constants are stored in the large constants area and their addresses similarly pushed on the stack.

OPTIMIZATION IN PASCAL/S

3.1: Introduction to Optimization in PASCAL/S

As stated previously the objective of this report is to document a ten-pass optimizing compiler. Three of the ten passes, in fact the three completely new passes, are designated as optimizing passes. Aho and Ullman [Aho 77a] give this description of code optimization:

> "Code optimization is a optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that does the same job as the original, but perhaps in a way that saves time and/or space."

In this description are several key points. Code optimization done in the PASCAL/S compiler is completely optional. (See figure 18.) To initiate optimization the programmer must specify which of the optimizing passes are to be included as part of the program's options. (See Appendix C.)

Code optimization is the process of rearranging and/or changing operations in a program during compilation so that a more efficient object program results. Code optimization in no way affects the outcome of the program. (See figure 19.)

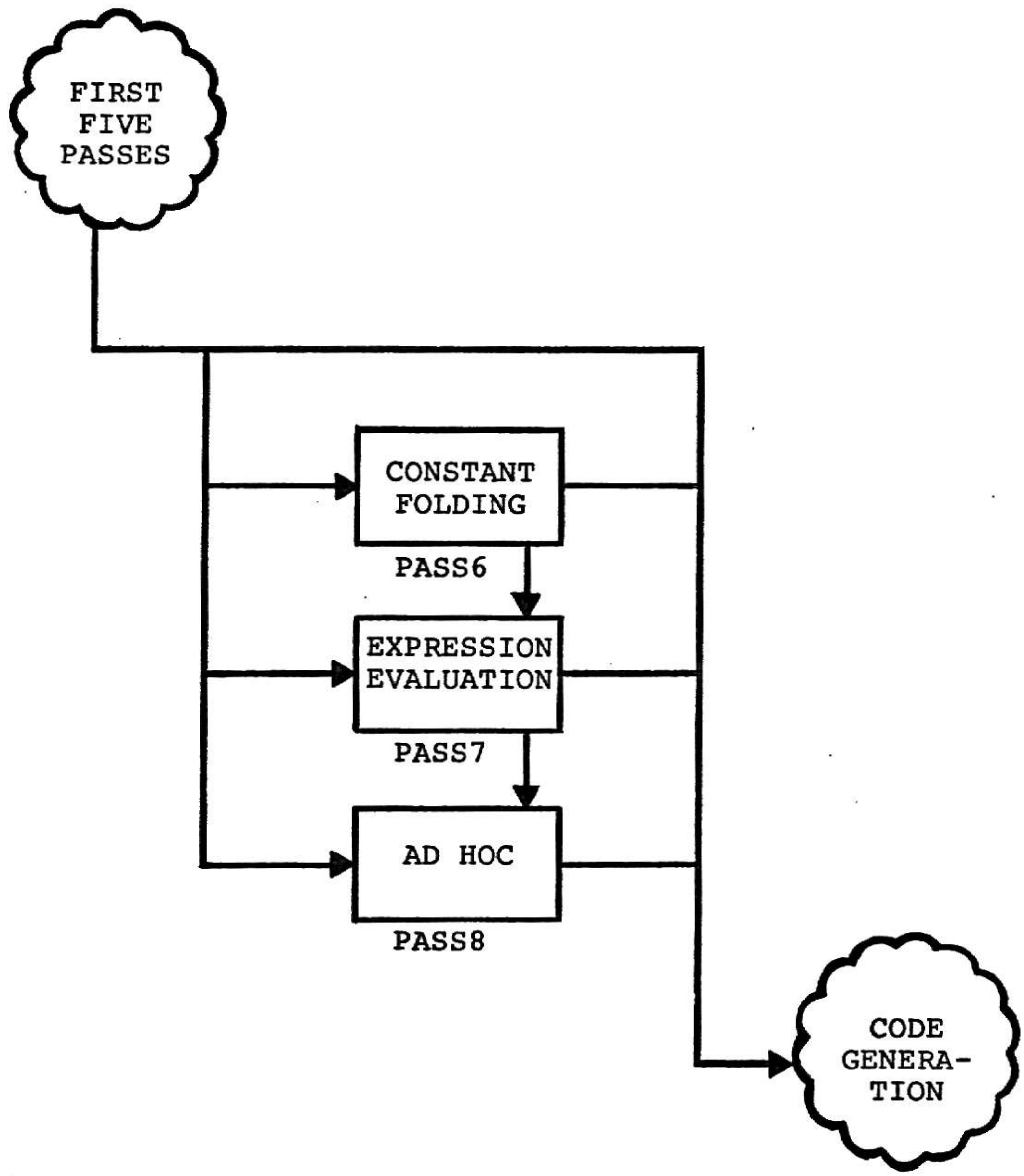
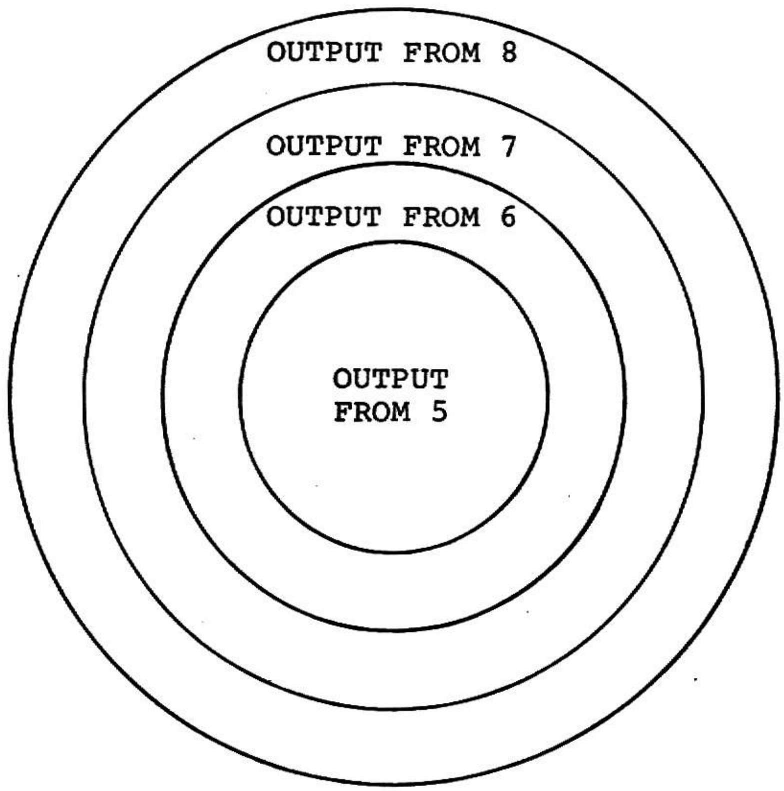


FIGURE 18: DATA FLOW OF OPTIMIZATION



The code output from each of the optimization passes is a superset of the previous pass. Generally speaking, the input to pass 9 is the output from 8 or any of the subsets.

FIGURE 19: OUTPUT CODE OF OPTIMIZATION

The basic reasons for including an optimization phase in the PASCAL/S compiler being developed are the following:

1. reduce the code size,
2. increase execution speed, and
3. minimize the number of temporary values on the stack.

The advantages follow quite logically. Reduced code size implies that larger programs can be run on smaller machines. Increased execution speed is always advantageous where possible. Minimizing the number of temporary values on the stack is perhaps the least obvious of the goals. The run-time storage area consists of two parts, a heap and a stack. The heap and stack originate from oposite ends of this area and grow toward each other. Temporary values are one of the principal entries on the stack. Therefore minimizing the number of temporaries decreases the size of the stack, which in turn leaves more space for dynamic allocation in the heap. (See figure 20.)

There exist two fast registers which contain the top two elements from the stack, first-in-stack (FIS) and second-in-stack (SIS). The remainder of the stack is stored in memory. Clearly, any operations which use only the top two elements in the stack will execute significantly faster than those which must access stack elements from memory.

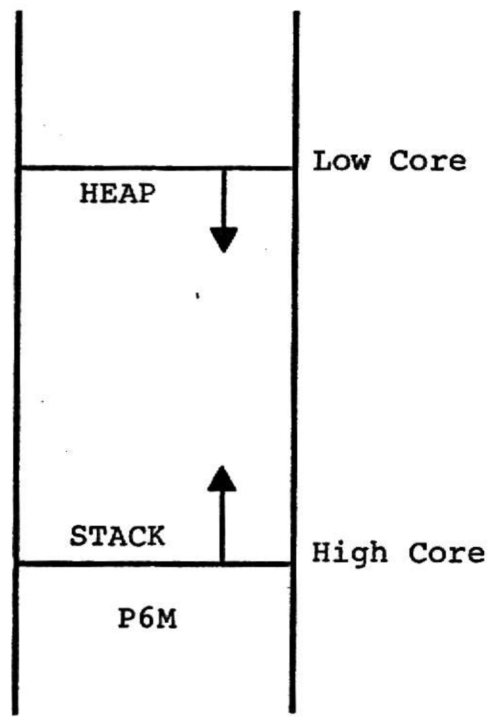


FIGURE 20: STACK & HEAP OF OPTIMIZATION

Thus all the attention paid to stack depth is in essence to concentrate on operations which access only the top two elements.

As with the rest of the compiler, each of the three optimization passes is well structured and concentrates on a specific aspect. These passes were described briefly in the introduction to the PASCAL/S compiler in Chapter 1. Once again, pass 6 is designed to do constant folding, pass 7 is designated expression evaluation, and pass 8 is an ad hoc pass which includes optimization misfits and eliminates invariant code.

The remainder of this chapter will consist of examinaing the code optimization of the PASCAL/S compiler by looking at each pass in some detail.

3.2: Constant Folding

Constant folding is the process of doing all operations at compile time which are static. Thus constant folding can be done at compile time with arithmetic, logical, or relational operators when the operand values are known or constant. This eliminates the need for executing these operations of the source program at run time. Thus, the optimization done in this pass primarily involves operations performed on constants.

Pass 6 accomplishes constant folding by using tree structures. These trees are built at the procedure or routine level only and constitute the complete structure of the procedure or routine. This allows for optimization not otherwise possible. Trees are built from the operands and operators of the intermediate code. With each block entry, new trees are built and pointers to the nodes, branches, or trees are saved in an operand stack. When a block exit is encountered, optimization is initiated on the trees of that block. Once completed, intermediate code is generated and the pointers to the trees of that block are popped off the stack and a new block entered.

Since the trees are not built at the program level, optimization is limited to within each routine. One other potential disadvantage is that of limited memory capacity, which is often the case with minicomputers. The tree representation of a long routine could exceed this limit. However, long routines are not justifiable from the point of view of good programming techniques.

Few changes are made to the form of the code processed in this pass. Much of the code is transferred directly to the output intermediate code just as it was input, and the remainder may involve deletion or rearranging and changing operations to produce more efficient object code.

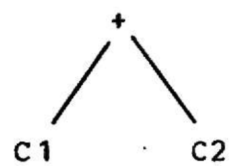
The remainder of this section is comprised of the specific techniques used in this pass and how and when they are applied. The description of each includes the type of optimization, an example in the form of a sentence segment, and mention of its contribution toward the goals of optimization.

In the remainder of this section, operands which begin with 'C' indicate constants, e.g. C1, C2, etc.; operands which begin with 'BE' represent Boolean expressions, e.g. BE1, BE2, etc.; and operands which begin with 'RE' represent relational expressions, e.g. RE1, RE2, etc.

- constant folding on binary arithmetic or relational operators

Consider the addition of two constants C1 and C2. The code generated without optimization is as follows:

```
PUSHCONST  (C1)
PUSHCONST  (C2)
ADD
```



During this pass this code would be replaced by the single instruction:

```
PUSHCONST    (C3)      where C3 = C1 + C2
```

This means a reduction in code size, faster execution and the savings at run time of a temporary on the stack.

- constant folding on unary operators

This aspect of constant folding applies to such operators as unary minus, abs, chr, and ord. In the unoptimized version, the code generated would be:

PUSHCONST (C1)

OP

When folding is done the resulting code is:

PUSHCONST (C2) where C2 = op(C1)

Again, code is eliminated and an increase in execution speed is gained.

constant folding of Booleans

Consider the following examples:

a) FALSE AND <BOOLEAN EXPRESSION>

b) TRUE OR <BOOLEAN EXPRESSION>

In a) anytime the situation arises where a value of FALSE is to be 'ANDed' with a Boolean expression, there is no need to evaluate the Boolean expression. The existence of the FALSE implies that the result of the 'ANDing' will always be false.

Similarly for b), anytime a TRUE is 'ORed' with another Boolean expression, the result is always true, regardless of the value of the Boolean expression. Optimization of these expressions would result in

A) PUSHCONST (FALSE)

and

B) PUSHCONST (TRUE)

respectively.

With FALSE or <BE> and TRUE and <BE>, the 'false or' and the 'true and' are removed from the tree.

In this case there would be a reduction in code, an increase in execution speed and a potential savings of temporaries on the stack.

- constant folding of sets

To build a simple set such as [C1, C2, C3], the unoptimized code generated would be:

```
PUSHSET    [ ]       ( the null set is pushed
                       on the stack )
PUSHCONST  (C1)
INCLUDE
PUSHCONST  (C2)
INCLUDE
PUSHCONST  (C3)
INCLUDE
```

When all the values of the set elements are known at compile time, it is possible to optimize the code so that the resulting code is one instruction:

```
PUSHSET   [C1, C2, C3]
```

Clearly, code is eliminated which would contribute to faster execution, and the number of temporaries would also be reduced.

- constant folding related to indices

In the unoptimized version, the code generated for a subscripted element A[C1], is the following:

```
PUSHADDR   (A)
PUSH    (C1)
INDEX(min, max, element size)
```

In this sequence the address of A, the beginning address of the array, is pushed on the stack. Next the index, C1, is pushed on the stack. The 'index' instruction then looks at the FIS and checks to insure that it is within the range specified by the min and the max. If it is within the range, then the index (FIS) is multiplied by the length of an element in the array. The result, the offset, is then added to the starting point of the array (SIS) to arrive at the address of the element.

When the index is a constant, this may be done at compile-time eliminating the need for doing it at run-time. When folded during optimization the code generated is merely:

```
PUSHADDR    (B)
```

where B = (A + C1 * element size)

Amount of code decreases; execution speed increases; and the number of stack entries decreases.

- constant folding related to fields within records

Consider the code generated for a record R with a field E2 in the unoptimized version:

```
PUSHADDR    (R)
FIELD       (offset of element E2)
```

The field within a record is always a constant. As such it is possible to create the address at compile time rather than at run-time. The code in the optimized version would consist of the single instruction:

```
PUSHADDR    (R.E2)
```

where R.E2 = (R + offset of element E2)

It is not the case that this optimization may be performed with a field which is referenced within a BEGIN-END block as part of a WITH statement qualification. In this particular case the address of the beginning of the record does not immediately proceed the field, and therefore the starting point is not easily accessible at compile time.

The impact of this particular optimization is to reduce code generated and increase execution speed while decreasing the stack size.

- adding and subtracting zero and multiplying and dividing by one

All occurrences of adding or subtracting zero or multiplying or dividing by one are eliminated since they have absolutley no effect.

- zero divide check and overflow

This pass also checks for zero divide and overflow in approximately the same way as at run time. Such instances are flagged and error messages are produced.

- multiplication and division changed to shifts

The multiplication and division operations are changed to shifts, which result in faster exectuion.

- constant folding on real numbers

Constant folding for real numbers has not been included in this pass. This was not implemented primarily because of space limitations and objectives of the compiler.

Generally, constant folding for real numbers would be most helpful in the areas related to numerical analysis. Since this compiler is designed primarily for use in writing operating systems, there was not sufficient need to warrant implementation of constant folding on reals.

3.3: Expression Evaluation

Expression evaluation, the second of the optimizing passes, focuses on evaluating expressions in such a way as to increase execution speed or to reduce the stack size by minimizing the number of temporaries on the stack. It turns out that accomplishing these goals may actually in some cases increase the code size. For expression evaluation, once again trees are built from the operands and operators of the intermediate code and optimization is performed on the trees of each block.

The remainder of this section is devoted to the specific instances of optimization which are handled in this pass. The description will be structured to include the type of optimization, one or more examples, and the implications for optimization.

It is important to remember that the optimization being described by looking at sentences or sentence fragments (that is, expressions) may actually be performed repeatedly and at all levels in the trees which have been constructed and are being analyzed.

- Boolean expression evaluation:

a) Given the source code with two Boolean expression conjoined with 'and' (e.g. BE1 AND BE2), then the input code for this pass would be of the form

BE1 BE2 AND FALSEJUMP(lblx)

where lblx is still a mnemonic label rather than a displacement. It is possible, given this construct, to increase execution speed in some cases by modifying the code to be:

BE1 FALSEJUMP(lblx) BE2 FALSEJUMP(lblx)

Since an 'AND' requires that both arguments be true to yield a true result, it is unnecessary to evaluate the entire expression if the first Boolean expression (BE1) is false.

Note the length of the code is increased by the length of the false jump's argument (either 16, 24, or 32 bit displacement). In all but very extreme cases, this would be 16 bits.

b) Given the source code with two Boolean Expressions conjoined with 'OR' (e.g. BE1 OR BE2), then the input to this pass would be of the form:

BE1 BE2 OR FALSEJUMP(lblx)

Similar to the previous example, it is possible to increase execution speed by modifying the code to be:

BE1 TRUEJUMP(lbly) BE2 FALSEJUMP(lblx) DEFLABEL(lbly)

This is possible since the 'OR' operator requires only one of its operands to be true to produce a true result. When the first expression is evaluated TRUE, the truejump branches around the remainder of the expression to the operator/operand which follows the false jump in the code stream. The remainder of the expression evaluation works as described previously. Again, the length of the code is increased by a displacement length.

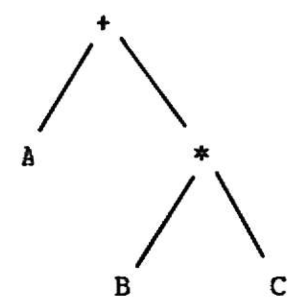
- **operand switching**

This aspect of optimization is not well documented in the literature. Generally though, the objective here is to reduce the number of temporary registers used in the expression evaluation. However, within the PASCAL/S version, the objective is to reduce the stack size.

a) arithmetic operators: For an example of operand switching with arithmetic operators, consider the following arithmetic expression:

A + B * C

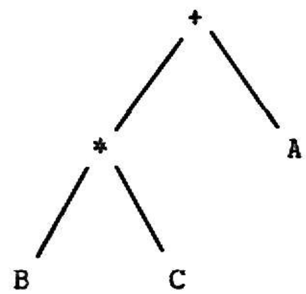
which yields the following tree:



Thus, the code generated in an unoptimized version would be:

```
PUSHVAR    A
PUSHVAR    B
PUSHVAR    C          <------ stack depth = 3
MUL
ADD
```

To optimize the evaluation of this expression the operands must be 'switched'. In essence, the branches of a node shall be switched so that the longest (that is, the branch requiring the most number of entries on the stack) subtree is on the left branch. In this case the new tree would be:



and the code generated in the optimized version would be:

```
PUSHVAR    B
PUSHVAR    C       <-------  stack depth = 2
MUL
PUSHVAR    A       <------- stack depth = 2
ADD
```

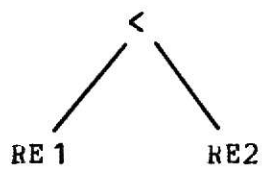
Obviously, operand switching on arithmetic operators is limited to additon and multiplication, that is the arithmetic operators which are commutative by definition. It is interesting to note that, in addition to reducing the size of the stack, this optimization results in the operands always being in the fast access portion of the stack (FIS and SIS).

b) relational operators: With respect to relational operators, it is sometimes necessary to modify the operators when operand switching is done. Consider the expression:

RE1 < RE2

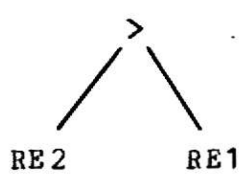
where the absolute stack lengths are such that the length of

RE1 is less than RE2 (i.e. RE > E1).



In order to minimize stack depth the relational expression, RE2, should be on the left branch. Thus, optimization in the form of operand switching is required. The results would be:

RE2 > RE1



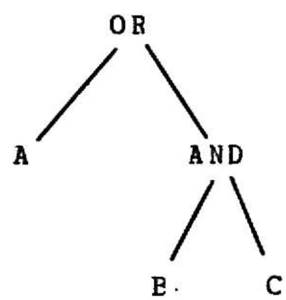
The impact of this optimization is to reduce the stack depth and keep the operands in fast registers by evaluating the longest branch first. (Note: The size of the code remained the same.)

The relational operators of >, >=, <=, < all require the switching of operators when the operands are switched. The relational operators of <> (not equal) and = do not necessitate the switching of the operators when their operands are switched.

c) Boolean operators: Consider the following expression segment as an example related to Boolean operators:

A OR (B AND C)

with tree:



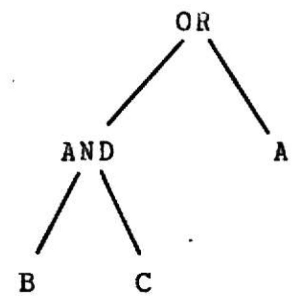
and unoptimized code:

```
PUSHVAR     (A)
PUSHVAR     (B)
PUSHVAR     (C)          <-----stack depth = 3
AND
OR
```

when optimized with operand switching, the new eqivalent expression is:

(B AND C) OR A

The revised tree is:



The optimized code is:

```
PUSHVAR    (B)
PUSHVAR    (C)
AND
PUSHVAR    (A)
OR
```

Following this operand switching, the Boolean expression optimization would proceed so that the code stream of

B C AND A OR FALSEJUMP(lblx)

would become

B FALSEJUMP C TRUEJUMP A FALSEJUMP <truepart> <falsepart>

- redundant operators

The circumstance of redundant operators might arise with two unary 'NOT' operators preceding a Boolean expression or two MINUS operators preceding an arithmetic expression (e.g. 'NOT NOT BE1').

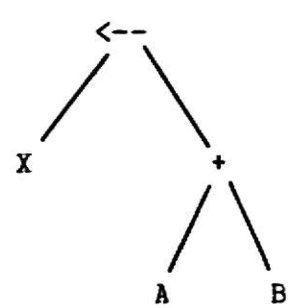
Optimization in this pass would eliminate these redundant operators since they have no impact on either the Boolean expression or the arithmetic expression respectively.

- **immediate operators**

Expression evaluation is extended further with 'immediate operand optimization'. Consider the following assignment statement:

X <-- A + B

where the tree consists of:

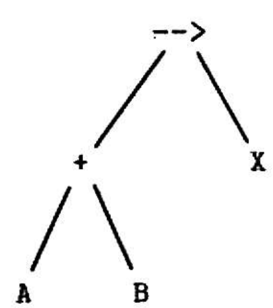


and the unoptimized code is:

```
PUSHADDR  X
PUSH  A
PUSH  B        <-------stack depth = 3
ADD
ASSIGN
```

With the optimization discussed thus far, the code and tree become:

```
PUSH  A
PUSH  B        <-------stack depth = 2
ADD
PUSHADDR  X
'SWAP' ASSIGN
```



Normally, the assign instruction would take the FIS and assign it to the address at SIS. When operand switching is done, however, these two are reversed. Rather than actually doing the PUSHADDR and 'SWAP' ASSIGN shown in the optimized code, there exists an instruction which allows the immediate assignment of the FIS to the address from within the code stream (pop FIS to INT). When this immediate type operator is used, the key is that the code (in the example the address of X) immediately follows the operator in the code stream.

Thus the code becomes:

```
PUSHVAR     (A)
PUSHVAR     (B)
ADD
ASSIGNIMM     (ADDR OF X)
```

These 'immediate' type operators are especially applicable to constants within arithmetic and relational expressions. For example, consider the arithmetic expression with variable A and constant C1:

A + C1

Normally, the code generated would be:

```
PUSH  A
PUSH  C1
ADD
```

By taking advantage of the instructions within the instruction set, it is possible to condense this code to:

```
PUSH  A
ADDIMM  C1
```

Consider also, the relational expression with variable A and constant C1:

```
A > C1
```

which generates unoptimized code:

```
PUSH A
PUSH C1
COMPARE
```

Following optimization applying the 'OP immediate' instruction the optimized code is:

```
PUSH A
COMPAREIMM   (C1)
```

Immediate operators could be similarly applied to Booleans where at least one operand is a constant. However, this is accomplished in pass 6 by folding Boolean constants.

Thus, there is some savings in code as well as an increase in execution speed and a reduced stack size when 'op immediate' instructions are used. This approach can be applied to all operators when they operate on a constant or a simple single variable.

Once again, it is important to note that any of the specific instances of optimization discussed in this section may be applied at any level of any of the trees created for the block.

3.4: Ad Hoc or Invariant Optimization

The final pass of optimization includes diverse optimization features which do not fall appropriately into the categories of optimization of the previous two passes. Optimization is done on control structures to the extent that unreachable code is removed. The following examples illustrate situations which are representative of such optimization:

- IF FALSE THEN A:= E1 ELSE A:= E2;

  can be optimized to: A:= E2
- IF TRUE THEN A:= E1 ELSE A:=E2

  can be optimized to: A:= E1
- CASE C1 OF . . .

  can be optimized to include only the proper one from the enumeration
- WHILE FALSE DO . . .

  can be deleted
- REPEAT <statements> UNTIL FALSE

  can be optimized to: <statements>

Optimization which removes invariant code from loops is also done. That is, operators whose operands are unaffected by code within the loop may be removed from the loop itself and thus execute once versus the numerous times when inside the loop.

Optimization is done which removes all branches to branches. Further, all branches are flagged to insure that they can be reached.

This pass performs factoring of subexpressions. For example:

```
-C * ( - (A + B) + D)  =>
-C (- ((A + B) - D) =>
 C * (A + B - D)
```

This pass scans for identical operations within each block. Thus if the following two statements occurred in the same block, the temporary, resulting from evaluating A * B is calculated only once and saved on the stack.

```
X := X + (A * B)
Y := Z * (A * B)
```

Another major contribution of this pass is strength reduction, which replaces an expensive (in time) operation by a cheaper one. Often this means multiplication vs. addition respectively.

The following example involves array subscripts which are the most representative of strength reduction:

```
FOR I:= A TO B DO X[I]:= E
```

where the array element is calculated by:

```
PUSHADDR    (X)
PUSHVAR     (I)
INDEX(min, max, length)
```

can be optimized as follows:

```
PUSHADDR    (X)
PUSHVAR     (A)
INDEX(min, max, length)
<FOR . . .>
PUSHADDR    (T1)
PUSHCONST    (length)
ADDIND
PUSHADDR    (T1)
PUSHVAR     (E)
STORE
```

In this example, the code size is increased; however execution speed will decrease. In this instance optimization requires trade-offs.

CODE GENERATION

4.1: Introduction

The PASCAL/S compiler's last two passes are dedicated to code generation, using the common two-pass design. PASCAL/S pass 9 performs code selection while pass 10 does code assembly.

In effect, code generation must take the intermediate code input and convert it into the object program. (See figure 21.) Aho and Ullman state that "designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design, both practically and theoretically" [Aho 77a]. Good code generation dictates utilizing the facilities of the hardware as effectively as possible, which is difficult to do in an optimal way.

One reason that code generation is difficult is a result of its dependence upon particular machines. Thus, code generation must produce an object program which performs the computations directed by the intermediate code input, by efficiently using the instruction set of the machine.

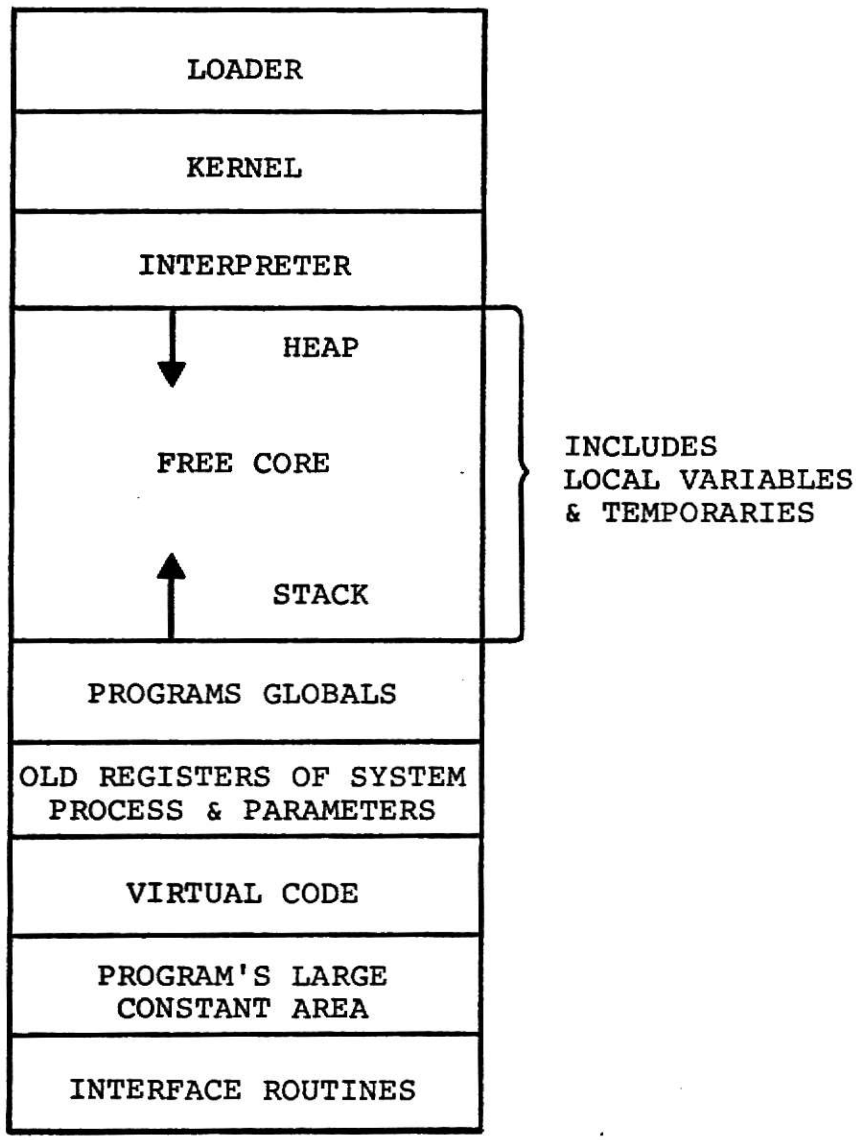


FIGURE 21: ENVIRONMENT OF THE PASCAL MACHINE

The remainder of this chapter contains discussion of code generation by first describing code selection and then code assembly.

4.2: Code Selection

Pass 9 of the PASCAL/S compiler performs code selection and is similar in task to Hartmann's pass 6. The main distinction between the two, is in the code which is being selected. The code selected for PASCAL/S is the code for the stack machine being developed by Perkin-Elmer Data Systems [Goet77a]. Hartmann's compiler, however, performed code selection for Brinch Hansen's virtual Pascal machine (in essence the kernel and interpreter written by Brinch Hansen).

The tasks of this pass are to define the addresses of program labels, determine the run-time stack requirements of routines, pull constants out-of-line and place them in a constants table, and translate the intermediate code input into the proper object code.

Before discussing the specifics of code selection, it must be noted that the PASCAL/S instruction set is quite exhaustive and differs from instruction sets associated with machines of the Von Neumann architecture. The PASCAL/S instruction format consists of an 8 bit profile, followed by an 8 bit op code, followed by zero or more arguments (operands).

The profile specifies the base register to be used with the operator, the length of the operands -- which is either known implicitly by the instruction op code or is specified -- and the length of any displacement.

The extensive instruction set when combined with the profile allows for efficient code generation. The instruction set is obviously oriented toward the stack machine for which it is designed. This instruction set includes many virtual instructions (e.g. flow-of-control) similar to those used within Brinch Hansen's virtual Pascal machine (e.g. ENTERMON, ENTERPROCESS, kernel call, etc.).

'Selecting' the proper object code is done by first analyzing the types of operands which indicate the object code operator. Next, the mode of any address, the length of any operands and the length of any displacement are determined. This information thus allows the proper profile to be created and hence the proper code to be selected.

This pass creates four tables -- a routine label table, a jump table, a stack table and a constants table which are to be used by the next pass. These tables contain the addresses of routine labels,the addresses of jump labels, the stack requirements of routines, and the large constants (string constants only in PASCAL/S) respectively.

Information which is used in code selection comes from several of the previous passes. Body analysis passes along the number of routine labels, the number of jump labels, and the length of the constants area through the inter-pass record. The number of routine labels was determined during declaration analysis. Syntax analysis recorded the number of jump labels. The length of the constants area was determined by name analysis. This size information is used by code selection to allocate tables during the initialization of the pass.

One of the functions of this pass is to define the addresses of program labels which are either routine labels or jump labels. Routine labels occur at the beginning of each routine body, one per routine. Routine labels are delineated during declaration analysis of routine declarations. Jump labels only occur within routine bodies. Jump labels are created during syntax analysis as statements are converted to postfix notation.

In the input code, routine labels appear as arguments to the ENTER command, which begins all routine bodies. When such a label is encountered, the current program address is added to the routine label table. The entries from this table are used in the next pass to replace the labels of the call instructions by addresses.

In the input code, jump labels are represented by a label command followed by a label number. When such a label is encountered the current program address is added to the jump label table indexed by the label number. As with routine labels, the next pass will use the information from the jump label table to replace the label in jump instructions with a relative address.

Jump labels are associated with four different instructions, the jump, the false jump, the true jump, and the case jump. The first three are followed in the code by a specific label number. The case jump, however, is followed by the minimum and maximum case label 'values' and precisely maximum - minimum + 1 labels.

The displacement argument of jump commands may be one of three lengths D16 (16 bits), D24 (24 bits) or D32 (32 bits) depending upon the actual jump displacement. The provision for variable displacements is included for optimization. As a result, another table is constructed during pass 9, the **jump table.** All jump displacements are assumed to be 16 bits during code selection. This table is used to adjust the displacement argument upward as needed.

The following example demonstrates the need for adjustment:

```
LOC I
---
LOC X JUMP NAME 2
---
LOC Y DEF LBL NAME 1
---
LOC V JUMP NAME 1
---
LOC Z DEF LBL NAME 2
```

The displacement of the jump instruction at (a) is assumed to be 16 bits. The jump table is searched sequentially for each entry encountered, the name is looked up in the jump label table or the routine label table, and the difference between the current location in the jump table and the location of the label in the jump label table or routine label table forms the displacement. For the example, it is determined that D24 is required to represent this displacement. Thus, what was loc y should be adjusted to loc y + 1 along with all succeeding entries in the jump table. Note: The location counter is in bytes and there exists a one byte difference between these three displacement types. In the profile, the proper length displacement is updated from this table in the next pass.

Displacements into the variable area are static by this pass and their displacement lengths are known after pass 4. Even though they may be one of three lengths there is no need for concern in this pass.

Another function of code selection is determining the maximum run-time stack size of all routines. This is done by computing the requirements for each routine and then using the sum of the stack requirements for all routines as the worst possible case for the program. This can be rather misleading. Sequential Pascal permits recursion, yet this estimate of the maximum run-time stack ignores the effect of recursion. Further, this estimate of the worst possible case is highly unlikely, since it implies that all routines must have been called at one time. Thus, the stack requirements represented by this estimate might prohibit the execution of a program whose actual stack requirements were much less.

Input to code selection consists of approximately 50 unique commands. This pass encodes types into the opcodes, so that the command set is more than tripled (e.g. ADD now becomes either ADDINTFIS or ADDBYTEFIS, etc.). There are many extensions to the operand set of Brinch Hansen's which forms this exhaustive instruction set of the new machine.

Output from this pass lacks only the displacements for routine labels and jump labels to be the final machine code.

In summary, PASCAL/S pass 9 takes code which is oriented for the stack machine and selects the proper profile and operations for the PASCAL/S instruction set. It is quite similar to Hartmann's pass 6 and builds the same four tables, which remain in the heap for use by the succeeding pass.

4.3: Code Assembly

The final pass of the PASCAL/S compiler completes the transformation of the original program into its target program, the final machine code. The achievements of this pass are essentially the same as those of Hartmann's pass 7. Code assembly replaces routine labels and jump labels by their displacements. The maximum stack length for each routine is placed in the entry instructon of the routine. Error messages, if any, are written and the table of large constants is appended to the generated code.

The four tables created during code assembly are used by this pass. The addresses of labels are attained from either the jump label table or the routine label table and substituted for the labels, The entries of the stack table indicating the size of the run-time stack are used to complete the enter instructions for each routine. The table of constants is used so that the constants may be output following the code.

The listing of error messages is the remaining accomplishment of this pass. Errors have been flagged (with an error operator, the pass number, and the error message type) in previous passes as they were encountered. As a result, errors from the various passes will be listed in order of the line number. Code assembly is responsible for processing these error operators and printing the line number and the appropriate error message text.

Pass 10 optionally outputs the object code listing. This option may be specified by the programmer. (See Appendix C.) In conclusion, PASCAL/S pass 10 completes the translation of source code to final code.

COMMENTS AND CONCLUSIONS

This chapter is divided into a discussion of the results and impact of the project from two different perspectives: the author's personal point of view and the more general project view.

For the author, this was a satisfying venture. The documentation and code conversion tasks served to cultivate an understanding of the Sequential and Concurrent Pascal Compilers, and develop a thorough knowledge of the features added or modified. Creating documentation for a project of this size was indeed a new experience. This specific effort focused attention on facets of documentation essential in response to anticipated needs of others. Constructing the syntax graphs for the eleven languages associated with the new version of the compiler provided insight into the effect of each pass on the code produced. This project, as designed, was to modify and extend the existing compilers. This resulted in the author gaining a real appreciation for Pascal as an implementation language. The original compiler source text (supplemented by Hartmann's dissertation) was understandable and relatively easily modified.

The biggest benefit from the author's perspective was that this project was involved with state-of-the-art software development. Thus this project was considerably more satisfying than any of the class-type projects the author had been exposed to previously. The author is appreciative of the opportunity and enthusiastic about pursuing this area even more.

Related to the general project point of view the discussion will focus on the following areas: project expectations and effort, problems encountered, and notions about further development.

The focus of this applied research was to develop Pascal compilers, both Sequential and Concurrent, for a stack machine. The particular machine was to emulate the PASCAL/S machine which is being developed by a division of Perkin-Elmer Data Systems. The project included:

- modifying the Sequential and Concurrent Pascal compilers to produce PASCAL/S object code,
- writing an interpreter and modifying the kernel of the current KSU implementation [Neal76a] to emulate the new architecture,
- writing new passes to perform optimization,
- testing and debugging the compilers and the interpreter, etc., and
- documenting the modifications to Hartmann's Sequential compiler.

This research project was undertaken by a research team consisting of a faculty member and two part-time graduate students at Kansas State UniversiY. It is estimated that the total time spent working on the project by the entire team was over 2800 hours. The following accounting of time is based upon the best estimates and notes of the research team.

	approx. %	hours
Gary: (100% - 6 mos.)		
administrative tasks	10	164
learning	20	328
code conversion & writing	20	328
testing & debugging	50	820
	----	----
TOTAL	100	1640
Dave: (50% - 6 mos.)		
learning, design, implementation, & testing of interpreter	78	390
adapted Hartmann's intermediate operators to PASCAL/S operators (pass 9)	10	50
adapted kernel and Pascal loader module	12	60
	----	----
TOTAL	100	500
Barb: (50% - 4 mos.)		
learning	27	200
documentation		
. writing, editing, etc.	18	130
. syntax graphs	7	50
. Sequential Report	2	15
. editing & final report (since Sept. 19)	10	70
code conversion, writing, & debugging (CPascal - passes 1 - 5 and pass 1 XREF)	36	265
	----	----
TOTAL	100	730

There were several types of problems which were encountered during work on this project. Basically these problems resulted from the initial decision regarding the working copy of the compilers, machine down-time and availability, and an under-estimation of the effort required for project completion.

Early in the development process, the decision was made to use the Navy's version of the compilers, both Sequential and Concurrent, as the working copy. Initially it was felt that this would allow the research team to capitalize on some of the optimization flags and features which had already been incorporated in the Hartmann compiler. Unfortunately, this choice caused problems and cost time. Since it was also decided to model the Hartmann compiler as closely as possible, it would have saved effort to have started with Hartmann's compiler and then added the flags and features incorporated in the Navy's version which were related to optimization. Initially, those involved were unaware of the profound impact of some of the changes the Navy made related to I/O, parameter passing, data save areas, etc.

Machine down-time had a substantial negative impact on the research project. It was particularly a factor during the final weeks of the project when there were two weeks of intermittant up-time and two weeks of complete down-time. Most down-time resulted from problems with peripherals. Other incidences of machine down-time were frequent enough to be inconvenient and annoying.

Perhaps an even bigger factor was machine availability. Due to the heavy research load currently on the Interdata 8/32, machine availability was limited and the response-time was slow.

The original estimate of the time required to complete the project was closer to 1900 hours than the 2800+ hours actually reported. As a result, it would have helped considerably to have had more time to complete the current project. The division of responsibilty (labor and talents) seemed consistent with the original intent of the project, so that adding additional personnel probably would not have contributed toward on-time completion. An additional month in the overall plan would likely have guaranteed timely completion, however.

It is likely that further development of the compiler would be beneficial. There are other optimization features which could be added to improve the object code still further. Thus, there still exists opportunities to continue to improve upon the current version.

ANONOTATED BIBLIOGRAPHY

References to bibliographic items are sorted and indexed by a key which has the following format:

[<NAME> <YEAR> <LETTER>]

Where <NAME> is the first four letters of the last name of the senior author, <YEAR> is the year the article was published, and <LETTER> is a letter appended to the year which uniquely qualifies a paper if the author published more than one paper in a specific year.

[Aho 77a] Aho, A. and Ullman, J. Principles of Compiler Design, Addison Wesley, 1977.
An introductory text on compiler design with emphasis on solutions to problems commonly encountered in compiler design.

[Ball76a] Ball, M., Cottel, D. and Zaun, J. Concurrent Pascal, Navy Undersea Center, December, 1976.
This document describes the current implementation in use at the Navy Ocean Systems Center of Concurrent Pascal. Additionally, documentation is included for the various routines written specifically to handle I/O devices for the Interdata 7/16.

[Brin72a] Brinch Hansen, P. Structured Multiprogramming, Communications of the ACM 15, 7 (July, 1972), pp. 574-577.
This paper develops a structured representation of multiprogramming in a high level language. Included is a combination of contitional critical regions and event variables which allow control of resource scheduling among competing processes.

[Brin73a] Brinch Hansen, P. Concurrent Programming Concepts, Computing Surveys, vol. 5, no. 4, December, 1973.
This paper describes multiprogramming features including event queues, semaphores, critical regions and monitors. Presented are two principles for guiding the choice among these language concepts. Annotated algorithms are used to illustrated the basic problems associated with multiprogramming.

[Brin74a] Brinch Hansen, P. Universal Types in Concurrent Pascal, Information Science, California Institute of Technology, November, 1974.
The third of three papers in 'Concurrent Pascal Introduction', describes Universal types as implemented in Concurrent Pascal. Universal types imply that type checking of operands may be relaxed as needed in system programming.

[Brin75a] Brinch Hansen, P. The Programming Language Concurrent Pascal, Information Science,

California Institute of Technology, February, 1975.
The first of three papers in 'Concurrent Pascal Introduction' describes concurrent tools, processes and monitors which extend the programming language Sequential Pascal. Concepts are explained via illustrations and an example. The monitor concept within Concurrent Pascal is such that the hierarchy of access rights to shared data can be explicitly stated in the program and checked by the computer.

[Brin75b] Brinch Hansen, P. Job Control in Concurrent Pascal, Information Science, California Institute of Technology, March, 1975.
This paper is the second of three papers included in 'Concurrent Pascal Introduction'. This paper describes how a Concurrent Pascal operating system when started prempts Sequential Pascal programs. Also in the paper are a description of using Sequential Pascal as job control language and the interaction of Sequential Pascal program with the operating system.

[Brin75c] Brinch Hansen, P. Concurrent Pascal Report, Information Science, California Institute of Technology, June, 1975.
This report defines the programming language for structured programming of operating systems--Concurrent Pascal. The report includes compelte discussion of syntax and the extensions beyond Sequential Pascal of system types--processes, monitors and classes.

[Brin75d] Brinch Hansen, P. Disk Scheduling at Compile Time, Information Science, California Institute of Technology, June, 1975.
This paper presents a simple algorithm for allocating program files on a moving head disk. The method described claims to combine the best features of consecutive and non-consecutive allocation, fast sequential access and fast allocation. This method is used in PBH's SOLO operating system.

[Brin75e] Brinch Hansen, P. The Programming Language Concurrent Pascal, IEEE Transactions on Software

Engineering 1, 2, June, 1975.
A collection of three papers annotated in [Brin75a], [Brin75b] and [Brin74a].

[Brin75f] Brinch Hansen, P. Sequential Pascal Report, Information Science, California Institute of Technology, July, 1975.
This report defines the programming language Sequential Pascal as implemented for the PDP 11/45 computer.

[Brin75g] Brinch Hansen, P. The SOLO Operating System: A Concurrent Pascal Program, Information Science, California Institute of Technology, July, 1975.
The first of three papers in 'The SOLO Operating System', this paper describes SOLO--a single user operating system-- which is written in Concurrent Pascal. SOLO utilizes the abstract data types (classes, monitors and processes) such that most access rights are checked at compile-time. The paper describes SOLO from the user's point of view by describing the processes and the interaction of sequential programs with the system.

[Brin75h] Brinch Hansen, P. The SOLO Operating System: Job Interface, Information Science, California Institute of Technology, July, 1975.
The second of three papers in 'The SOLO Operating System', this paper describes the interface between the SOLO Operating system and Sequential Pascal programs.

[Brin75i] Brinch Hansen, P. The SOLO Operating System: Processes, Monitors and Classes, Information Science, California Institute of Technology, July, 1975.
The third of three papers in 'The SOLO Operating System' describes SOLO as implemented in Concurrent Pascal. The overall structure and the details are described.

[Brin75j] Brinch Hansen, P. The Concurrent Pascal Compiler, Information Science, California Institute of Technology, October, 1975.
The fourth of four papers in 'Concurrent Pascal Machine', this paper describes the

Concurrent Pascal Compiler, which is a portable, seven pass compiler written in Sequential Pascal and requires 16K words of core store. Each pass is summarized along with its interface to the operating system.

[Brin75k] Brinch Hansen, P. Concurrent Pascal Machine: Store Allocation, Information Science, California Institute of Technology, October, 1975.
The first of four papers in 'Concurrent Pascal Machine' describes the scheme of allocation of core store among the various processes of a Concurrent Pascal program.

[Brin75l] Brinch Hansen, P. Concurrent Pascal Machine: Code Interpretation, Information Science, California Institute of Technology, October, 1975.
The second of four papers in 'Concurrent Pascal Machine', this paper describes the process of interpretation of the virtual code generated by the Concurrent Pascal Compiler as accomplished on the PDP 11/45.

[Brin75m] Brinch Hansen, P. Concurrent Pascal Machine: Kernel, Information Science, California Institute of Technology, October, 1975.
The third of four papers in 'Concurrent Pascal Machine', this paper describes the kernel of Concurrent Pascal as implemented on the PDP 11/45 computer. Also described in the paper are the kernel control of processor multiplexing and the scheduling of monitor calls.

[Brin75n] Brinch Hansen, P. A Real-Time Scheduler, Information Science, California Institute of Technology, November, 1975.
This paper describes a real-time scheduler for process control applications given a fixed number of tasks which are to be carried out periodically as determined by the operator. The design, programming and testing of the program are described in detail.

[Brin76a] Brinch Hansen, P. Concurrent Pascal: Implementation Notes, Information Science, California Institute of Technology, January, 1976.

This paper contains information about porting the SOLO operating system and the programming language Concurrent Pascal to other computers. This paper is technical enough to require familiarity with other PBH papers.

[Brin76b] Brinch Hansen, P. The Job Stream System, Information Science, California Institute of Technology, January, 1976.

This paper describes an operating system which compiles and executes short user programs as input from a card reader and output to a line printer. This model of an operating system is written in Concurrent Pascal and uses buffers stored on disk to allow simultaneous input, execution and output.

[Brin77a] Brinch Hansen, P. The Architecture of Concurrent Programs, Prentice-Hall, 1977.

This book describes a systematic way of creating concurrent programs in Concurrent Pascal. The use of the language is illustrated in three non-trivial concurrent programs, including a single-user operating system. This book is particularly useful as a practical supplement in operating system courses.

[Goet77a] Goettelman, J. KSU Project: Definition/ Work Statement, Perkin-Elmer Data System Memorandum, January, 1977.

This memorandum provides a description of the problem and the expectations associated with KSU personnel's response to the problem including time constraints and final product.

[Grie71a] Gries, D. Compiler Construction For Digital Computers, John Wiley & Sons, Inc., 1971.

This book provides an introduction to compiler construction. Included are numerous examples and discussions of many of the techniques and methods employed in compiler construction.

[Hank77a] Hankley, W. and Rawlinson, J. Sequential Pascal Supplement for Fortran Programmers: A Primer of Slides, Kansas State University Department of Computer Science, Technical Report CS XX-XX, 1977.

This report consists of numerous slides designed to serve as an aid to introduce programmers familiar with Fortran to the version of Sequential Pascal currently implemented at KSU.

[Hart76a] Hartmann, A. A Concurrent Pascal Compiler for Mini Computers, Doctoral Dissertation, California Institute of Technology, 1976.

This Dissertation describes a seven-pass compiler for the Concurrent Pascal programming language. Each of the passes, written in Sequential Pascal, generates intermediate code. The final pass generates which can then be interpreted. This paper documents an implementatiof this compiler cn cn the PDP 11/45 computer at Cal. Tech.

[Hoar73a] Hoare, C.A.R. and Wirth, N. An Axiomatic Definition of the Programming Language Pascl, ACTA Informatica, 2, pp. 335-355, 1973.

This paper describes an implementation of Brinch Hansen's concept of a monitor for structured programming of operating systems. It describes a form of synchronization, the implementation in terms of semaphores, a proof rule and examples.

[Jens74a] Jensen, K. and Wirth, N. Pascal User - Manual and Report, Lecture Notes in Computer Science, 18, Springer-Verlag, 1974.

Primarily, this book is intended as a means of learning the programming language PASCAL for those already familiar with programming constructs.

[Lewi76a] Lewis, P., Rosenkrantz, D. and Stearns, R. Compiler Design Theory, Addison-Wesley Publishing Co. May, 1976.

This book discusses the mathemetical theory underlying the design of compilers and other language processors and describes how to use the theory in practical design.

[Neal76a] Neal, D., Anderson, G., Ratliff, J. and Wallentine, V. KSU Implementation of Concurrent Pascal -- A Reference Manual, Kansas State University Department of Computer Science, Technical Report CS 76-16, January 1, 1977.

This report reflects the porting of the language Concurrent Pascal to the Inderdata 8/32 at KSU. It is intended to serve as 'an overview to the implementation approach', 'a reference manual for the SOLO user on the 8/32', 'a reference manual for the Sequential Pascal Programmer using SOLO', and 'a configuration guide to SOLO systems maintenance personnel.'

[Nori74a] Nori, K.V., Ammann U., et. al. The Pascal 'P' Compiler: Implementation Notes, Institute for Informatika, Technical University, Zurich, Switzerland, December, 1974.

<>

[Wall76a] Wallentine, V. and McBride, R. Concurrent Pascal--A Tutorial, Kansas State University Department of Computer Science, Technical Report CS 76-17, November, 1976.

This paper provides several examples of the utility of Concurrent Pascal applications of the language. The examples include such applications as priority scheduling of resources, message system, the data base reader/writer problem, data link control procedures and network inter-process communication systems.

[Wels72a] Welsh, J. and Quinn, C. A Pascal Compiler for ICL 1900 series computers, Software Practice and Experience, 2, pp. 73-77.

<>

[Wirt71a] Wirth, N. The Programming Language Pascal, ACTA Informatica 1, 1, 1971, pp. 35-63.

<>

[Wirt75a] The Design of a Pascal Compiler, Software Practice and Experience 1, 1971, pp. 309-333.

<>

[Wirt75b] Wirth, N. An Assessment of the Programming Language Pascal, IEEE Transactions on Software Engineering 1, 2, June, 1975.

A brief assessment of Pascal is presented which includes an enumeration of features in constructing correct programs which have proven valuable without jepodarizing conceptual simplicity or efficient use of the language. Also, it includes discussion of features which may be controversial.

APPENDIX A:

MODIFIED SEQUENTIAL PASCAL REPORT

SEQUENTIAL PASCAL REPORT*

Per Brinch Hansen
Alfred C. Hartmann

Information Science
California Institute of Technology

July 1975

Original Abstract

This report defines the sequential programming language Pascal as implemented for the PDP-11/45 computer.

Current Abstract

This report, as modified, defines the sequential programming language Pascal as implemented for the Future System Architecture of Interdata Corp. and for a current version of the compiler on the Interdata 8/32 computer at Kansas State University.

Key Words and Phrases: Pascal, programming languages.

CR Categories: 4.2

* The changes in this report reflect changes in Hartmann's Compiler which were made by Gary Anderson. These changes were documented and added to this report by Barbara K. North.

CONTENTS

1. INTRODUCTION

1. Introduction

This report defines the sequential programming language Pascal implemented on the PDP-11/45 computer. Pascal is a general purpose language for structured programming invented by Niklaus Wirth. **Also included are modifications and changes within the language which are currently implemented on the Interdata 8/32. Changes to the original report are flagged by bold face type.**

This is a brief concise definition of Pascal. A more informal introduction to Pascal is provided by the following reports:

Wirth, N. Systematic Programming, Prentice-Hall, 1973.

Jensen, K. and Wirth, N. Pascal-User Manual and Report, Lecture Notes in Computer Science 18, Springer-Verlag, 1974.

The central part of this report is a chapter on data types. It is based on the assumption that data and operations on them are inseparable aspects of computing that should not be dealt with separately. For each data type we define the constants that represent its values and the operators and statements that apply to these values.

Sequential Pascal has been implemented for the PDP-11/45 computer at Caltech **and the Interdata 8/32 at Kansas State.**

2. Syntax Graphs

The language syntax is defined by means of syntax graphs of the form:

```
while statement
-->WHILE -->expr -->DO -->statement-->
```

A syntax graph defines the name and syntax of a language construct. Basic symbols are represented by capitals and special characters, for example

WHILE DO + ;

Constructs defined by other graphs are represented by their names written in small letters, for example

expr statement

Correct sequences of basic symbols and constructs are represented by arrows.

3. Character Set

Pascal programs are written in a subset of the ASCII character set:

```
            character

                |---> graphic character --->|
            --->|                           |--->
                |---> control character --->|

            graphic character

            --------> special character -------->
                  |                          |
                  |--------> letter -------->|
                  |                          |
                  |-------->  digit -------->|
                  |                          |
                  |-------->  space -------->|
```

A graphic character is a printable character.

The special characters are

| " $ % & ' ( ) *   [ ]

, - . / : ; < = > ? @ ¬

The letters are

A B C D E F G H I J K L M N

O P Q R S T U V W X Y Z _

a b c d e f g h i j k l m

n o p q r s t u v w x y z

The digits are

0 1 2 3 4 5 6 7 8 9

control character

```
---> (: ---> digits ---> :) --->
```

A control character is an unprintable character. It is represented by an integer constant called its ordinal value (Appendix A). The ordinal value must be in the range 0..127.

digits

```
----------> digit ---------->
      |                 |
      |<----------------|
```

4. Basic Symbols

A program consists of symbols and separators.

symb

CON. NEXT

VOL.

The special sym

\+ - ( )

(.

They have fixed ıg constants and comments).

The word symbols are

ARRAY	BEGIN	CASE	CONST	DIV
DO	DOWNTO	ELSE	END	FOR
FORWARD	FUNCTION	IF	IN	MOD
NOT	OF	OR	PROCEDURE	PROGRAM
RECORD	REPEAT	SET	THEN	TO
TYPE	UNIV	UNTIL	VAR	WHILE
WITH				

They have fixed meanings (except within string constants and comments). Word symbols cannot be used as identifiers.

identifier

```
--->letter ---------------------->
               |                 |
               |<-- letter <--|
               |                 |
               |<-- digit  <--|
```

An identifier is introduced by a programmer as the name of a constant, type, variable, or routine.

identifiers

```
------> identifier ------>
   |                   |
   |<------- ' <------|
```

separator

```
---------------> space ------------------>
   |                                    |
   |----------> new line ----------->|
   |                                    |
   |----> " --> comment --> " ------>|
   |                                    |
   |----------> (*-->comment-->*) -->|
   |                                    |
   |---------->  --->comment--->  -->|
```

Two constants, identifiers, or word symbols must be separated by at least one separator or special symbol. There may be an arbitrary number of separators between two symbols, but separators may not occur within symbols.

A comment is any sequence of graphic characters enclosed in delimiting symbols. The exceptions to this statement are " , *) or when the symbol is the same as the one to delimit the end of the comment.

5. BLOCKS

5. Blocks

The basic program unit is a block:

block

```
---> declarations ---> compound statement --->
```

It consists of declarations of computational objects and a compound statement that operates on them.

declarations

```
  |<--const definitions<--|   |------------------->|
  |                       |   |                    |
--------------------------------->var declaration----->|
  |                       |                            |
  |<--type definitions<---|                            |
                                 |<----------------|   |
                                 |                 |   |
                          <-------routines<---------|
```

A declaration defines a constant, type, variable, or routine and introduces an identifier as its name.

compound statement

```
--->BEGIN-------->statement------->END--->
            |                 |
            |<----- ; <------|
```

A compound statement defines a sequence of statements to be executed one at a time from left to right.

6. Constants

A constant represents a value that can be used as an operand in an expression.

constant definitions

```
-->CONST----->identifier---> = --->constant--> ; ----->
        |                                          |
        |<-----------------------------------------|
```

A constant definition introduces an identifier as the name of a constant.

constant

```
------------> identifier ------------>
     |                            |
     |--> enumeration constant -->|
     |                            |
     |------> real constant ----->|
     |                            |
     |------> string constant --->|
     |                            |
     |-------> set constant ----->|
```

7. Types

A data type defines a set of values which a variable or expression may assume.

type definitions

```
-->TYPE----->identifier-----> = -->type--> ; ----->
         |                                     |
         |<------------------------------------|
```

A type definition introduces an identifier as the name of a data type. In general, a data type cannot refer to its own type identifier. A pointer type may however refer to a data type before it has been defined.

type

```
-----------> identifier ----------->
      |                         |
      |--> enumeration type -->|
      |                         |
      |--------> REAL -------->|
      |                         |
      |-------> SREAL -------->|
      |                         |
      |-----> array type ----->|
      |                         |
      |-----> record type ---->|
      |                         |
      |-------> set type ----->|
      |                         |
      |-----> pointer type --->|
```

Enumeration types and reals can only be operated upon as a whole. They are simple types.

Arrays, records, sets and pointer types are defined in terms of other types. They are structured types containing component types.

A data type that contains a pointer type is a list type. All other types are nonlist types.

An operation can only be performed on two operands if their data types are compatible (Section 9).

7.1. Enumeration Types

An enumeration type consists of a finite, ordered set of values.

enumeration type

```
----------------> CHAR ---------------->
    |                                  |
    |-----------> BOOLEAN ---------->|
    |                                  |
    |-----------> INTEGER ---------->|
    |                                  |
    |---->(---> identifiers --->)--->|
    |                                  |
    |-->constant--> .. -->constant-->|
```

The types char, boolean, and integer are standard enumeration types.

A non-standard enumeration type is defined by listing the identifiers that denote its values in increasing order.

A non-standard enumeration type can at most consist of 128 constant identifiers.

An enumeration type can also be defined as a subrange of another enumeration type by specifying its min and max values (separated by a double period). The min value must not exceed the max value, and they must be compatible enumeration constants (Section 9).

enumeration constant

```
------------> identifier ------------>
    |                               |
    |-----> char constant ------->|
    |                               |
    |----> boolean constant ----->|
    |                               |
    |----> integer constant ----->|
```

The basic operators for enumerations are:

:= (assignment)

< (less)

= (equal)

> (greater)

<= (less or equal)

<> (not equal)

>= (greater or equal)

The result of a relation is a boolean value.

An enumeration value can be used to select one of several statements for execution:

case statement

```
-->CASE-->expr-->OF-->labeled statements-->END-->
```

A case statement defines an enumeration expression and a set of statements. Each statement is labeled by one or more constants of the same type as the expression. A case statement executes the statement which is labeled with the current value of the expression. (If no such label exists, the effect is unknown.)

labeled statements

```
----->enumeration constant-----> : -->statement----->
  |                        |                        |
  |<---------- , <---------|                        |
  |                                                 |
  |<---------------------- ; <----------------------|
```

The case expression and the labels must be of compatible enumeration types, and the labels must be unique.

Integer case labels must be in the range 0..127.

The following standard functions apply to enumerations:

succ(x) The result is the successor value of x (if it exists).

pred(x) The result is the predecessor value of x (if is exists).

An enumeration type can be used to execute a statement repeatedly for all the enumeration values:

for statement

```
-->FOR-->identifier--> := -->expr------->TO----->|
                                   |        |    |
                                   |->DOWNTO->|  |
                                                 |
                     <--statement<--DO<--expr<----|
```

A for statement consists of an identifier of a control variable, two expressions defining a subrange, and a statement to be executed repeatedly for successive values in the subrange.

The control variable can either be incremented from its min value TO its max value or decremented from its max value DOWNTO its min value. If the min value is greater than the max value, the statement is not executed. The value of the control variable is undefined after completion of the for statement.

The control variable and the expressions must be of compatible enumeration types. The control variable may not be a constant parameter, a record field, a function identifier, or an array element (Sections 7.3, 7.4, 11). The repeated statement may not change the value of the control variable.

7.1.1. Characters

The type CHAR is a standard enumeration type. Its values are the set of ASCII characters represented by char constants:

char constant

---> ' ---> character ---> ' --->

The following standard function applies to characters:

ord (x) The result (of type integer) is the ordinal value of the character x.

The ordering of characters is defined by their ordinal values (Appendix A).

7.1.2. Booleans

The type BOOLEAN is a standard enumeration type. Its values are represented by boolean constants:

boolean constant

```
------> FALSE ------->
   |              |
   |----> TRUE --->|
```

where FALSE<TRUE.

The following operators are defined for booleans:

```
&     (and)
or
not
```

The result is a boolean value.

A boolean value can be used to select one of two statements for execution. It can also be used to repeat the execution of a statement while a condition is true (or until it becomes true).

if statement

```
-->IF-->expr-->THEN-->statement--->ELSE-->statement---->
                                 |                     |
                                 |-------------------->|
```

An if statement defines a boolean expression and two statements. If the expression is true, then the first statement is executed; else the second statement is executed. The second statement may be omitted in which case it has no effect.

The expression value must be a boolean.

<u>while statement</u>

```
--->WHILE--->expr--->DO--->statement--->
```

A while statement defines a boolean expression and a statement. If the expression is false, the statement is not executed; otherwise, it is executed repeatedly until the expression becomes false.

The expression value must be a boolean.

<u>repeat statement</u>

```
-->REPEAT----->statement----->UNTIL-->expr--->
          |                  |
          |<----- ; <----|
```

A repeat statement defines a sequence of statements and a boolean expression. The statements are executed at least once. If the expression is false, they are executed repeatedly until it becomes true.

The expression value must be a boolean.

7.1.3 <u>Integers</u>

The type INTEGER is a standard enumeration type. Using INTEGER in type definitions is synonmous with either SHORT-a 16 bit enumeration (-32768..32767), STANDARD-a 32 bit enumeration (-2147483648..2147483647) or LONG-a 64 bit enumeration [-2 TO 63..(2 TO 63)-1]. Type INTEGER is one of these three depending upon the defaults of the specific implementation. Its values are a finite set of successive whole numbers represented by integer constants.

<u>integer constant</u>

```
      |--> + --->|
      |          |
------>---------->digits--->
      |          |
      |--> - --->|
```

The following <u>operators</u> are defined for integers:

+ (plus sign or add)

- (minus sign or subtract)

* (multiply)

div (divide)

mod (modulo)

The result is an integer value.

The following <u>standard</u> <u>functions</u> apply to integers:

abs(x) The result (of type integer) is the absolute value of the integer) x.

chr(x) The result (of type char) is the character with the ordinal value x.

conv(x) The result is the real value corresponding to the integer x.

7.2. Reals

The standard type REAL consists of a finite subset of real numbers represented by real constants in 8 bytes. The short type real, SREAL, consists of a finite subset of real numbers represented by real constants in 4 bytes.

real constant

```
             |---------------->|       |-> + ->|
             |                 |       |       |
-->digits---> . -->digits-----> E ----------->digits---->
                           |           |       |        |
                           |           |-> - ->|        |
                           |                            |
                           |--------------------------->|
```

The letter E represents the scale factor 10.

The following operators are defined for reals:

:= (assignment)

< (less)

= (equal)

> (greater)

<= (less or equal)

<> (not equal)

>= (greater or equal)

+ (plus sign or add)

- (minus sign or subtract)

* (multiply)

/ (divide)

The result of a relation is a boolean value. The result of an arithmetic operation is a real value.

The following <u>standard</u> <u>functions</u> apply to reals:

abs(x) — The result (of type real) is the absolute value of the real x.

trunc(x) — The result is the (truncated) integer value corresponding to the real x.

7.3. Array Types

An array consists of a fixed number of components of the same type. An array component is selected by one or more index expressions.

array type

```
-->ARRAY-->(.----->enumeration type----->.)-->OF-->type-->
        |      |                    |       |
        |      |<------- , <-------|       |
        |                                   |
        |-> [ ---->enumeration type---> ] ->|
               |                    |
               |<------- , <-------|
```

The index types must be enumeration types. The component type can be any type. The number of index types is called the dimension of the array.

array component

```
--->variable-->(.----->expr----->.)---->
             |     |          |       |
             |     |<--- , <---|       |
             |                        |
             |-> [ ---->expr-----> ] ->|
                    |         |
                    |<-- , <---|
```

A component of an n-dimensional array variable is selected by means of its variable identifier followed by n index expressions (enclosed in brackets and separated by commas).

The number of index expressions must equal the number of index types in the array type definition, and the expressions must be compatible with the corresponding types.

The basic *operators* for arrays are:

:= (assignment)

= (equal)

<> (not equal)

The operands must be compatible arrays. The result of a relation is a boolean value.

A one-dimensional array of m characters is called a *string type* of *length* m. Its values are the string constants of length m:

string constant

```
---> ' ----->character-----> ' --->
          |               |
          |<--------------|
```

The ordering of characters defines the ordering of strings. A string must contain an even number of characters.

The following *operators* are defined for strings (in addition to those defined for all array types);

<	(less)
>	(greater)
<=	(less or equal)
>=	(greater or equal)

The operands must be strings of the same length. The result of a relation is a boolean value. Enumeration types cannot be defined within record types.

7.4. Record Types

A record consists of a *fixed part* and a *variant part*. One of these (but not both) can be missing.

```
        record type

        --->RECORD--->field list--->END


    field list

      |--------------->|
      |                |
    ---->fixed part------->variant part---->
                     |                 |
                     |------------------>|
```

The fixed part consists of fields of fixed types.

```
    fixed part

    ---->identifiers--> : -->type---->
     |                              |
     |<----------- ; <-------------|
```

```
    variant part

    -->CASE-->identifier--> : -->identifier-->OF---->variant--
                                                 |
                                                 |<--- ; <----
```

The variant part defines a *tag field* and one or more different sets of fields (called *variants*). Each possible variant is labeled by one or more constants. A record of this type can represent any one of the variants. The value of the tag field defines the chosen variant.

variant

```
--->labels---> : --->(-->field list-->)--->
```

labels

```
----->enumeration constant----->
   |                          |
   |<---------- , <-----------|
```

The tag field and the labels must be of compatible enumeration types, and the labels must be unique.

A non-standard enumeration type used as a tag field type can contain at most 16 constant identifiers.

Integer variant labels must be in the range 0..15.

A field of a record variable is selected by means of its variable identifier followed by the field identifier (separated by a period).

record component

```
--->variable---> . --->identifier--->
```

A variant field can only be selected if the value of the tag field is equal to one of the lables of that variant.

The basic operators for records are:

:= (assignment)

= (equal)

<> (not equal)

The operands must be compatible records. The result of a relation is a boolean value.

A with statement can be used to operate on the fields of a record variable:

with statement

```
--->WITH----->variable----->DO--->statement--->
          |               |
          |<---- , <----|
```

A with statement consists of one or more record variables and a statement. This statement can refer to the record fields by their identifiers only (without qualifying them with the identifiers of the record variables).

The statement

```
with v1, v2, ..., vn do S
```

is equivalent to

```
with v1 do
with v2, ..., vn do S
```

7.5. Set Types

The set type of an enumeration type consists of all the subsets that can be formed of the enumeration values:

```
set type

---->SET--->OF--->type--->
```

The component type of a set type is called its base type. It must be an enumeration type.

Set values can be constructed as follows:

```
set constructor

          |------------->|
          |              |
---->(.------>expr------->.)--->
 |          |       |         |
 |          |<-- , --|        |
 |                            |
 |-> [ ----->expr-----> ] ->|
            |       |
            |<-- , --|
```

A set constructor consists of one or more expressions enclosed in brackets and separated by commas. It computes the set consisting of the expression values. The set expressions must be of compatible enumeration types.

A set of integers can only include members in the range 0..127.

The empty set is denoted

(..)

The basic <u>operators</u> for sets are:

:= (assignment)

<= (contained in)

>= (contains)

- (difference)

& (intersection)

or (union)

The operands must be compatible sets. The result of a relation is a boolean value. The result of the other operators is a set value that is compatible with the operands:

in (membership)

The first operand must be an enumeration type and the second one must be its set type. The result is a boolean value.

7.5.1 Set Constants

The set constant consists of a finite binary string, which must be completely specified. A set constant may consist of a binary string of length 32, 64, or 128 only. A repetition factor may be used to facilitate completely specifying the binary string.

set constant

```
------> [ --> B --> ' -->bits--> ' ---> ] ------>
  |                                        |
  |--> (. --> B --> ' -->bits--> ' --> .) -->|
```

bits

```
   |<---------------------------------|
   |                                  |
------>bit---------------------------->|--->
   |                                  |
   |-->repetition factor-->bit-->|
```

repetition factor

```
------> ( --->integer constant---> ) ------>
```

bit

```
---->----> 0 ----->--------->
      |           |
      |--> 1 -->|
```

7.6. Pointer Types

A pointer type is a reference to another type:

pointer type

```
----> @ -----> type --> type ---->
 |         |
 |--> ¬ -->|
```

pointer component

```
                          |--> ¬ -->|
                          |         |
----> variable -----> @ ----->
```

The type referenced by a pointer is its component type. The component of a pointer variable is selected by means of its variable identifier followed by the symbol @ or ¬.

The basic operators for pointers are:

:= (assignment)

= (equal)

<> (not equal)

The operands must be pointers to compatible components.

An assignment associates the component of one pointer variable with another pointer variable as well.

Two pointers are equal if both are associated with the same component. The result of a pointer comparison is a boolean.

The pointer constant NIL denotes an undefined component. Initially all pointer variables have the value NIL. They may get a new value by assignment or by the standard procedure:

new(p) Associates a new component with the pointer variable p.

8. VARIABLES

8. Variables

A variable is a named store location that can assume values of a single type. The basic operations on a variable are assignment of a new value to it and a reference to its current value.

var declarations

```
-->VAR----->identifiers--> : -->type--> ; ---->
        |                                  |
        |<---------------------------------|
```

A variable declaration defines the identifier and type of a variable.

The declaration

var v1, v2, ... , vn: T;

is equivalent to

var v1: T; v2: T; ... ; vn: T;

variable

```
--->identifier-------------------------------->
                  |                        |
                  |<--array component<-----|
                  |                        |
                  |<--record component<----|
                  |                        |
                  |<--pointer component<---|
                  |                        |
                  |<--identifier<-- . <----|
```

A *variable* is referenced by means of its identifier. A *variable component* is selected by means of index expressions, field identifiers, or pointer references (Sections 7.3, 7.4, 7.6).

<u>assignment</u>

```
--->variable---> := --->expr--->
```

An assignment defines the assignment of an expression value to a variable. The variable and the expression must be compatible.

The variable may not be a constant parameter (Section 11).

9. Expressions

An expression defines a computation of a value by application of operators to operands. It is evaluated from left to right using the following priority rules:

First, factors are evaluated.

Secondly, terms are evaluated.

Thirdly, simple expressions are evaluated.

Fourthly, complete expressions are evaluated.

expr

```
-->simple expr-------------------------------------->
              |   |  |   |  |  |   |      |
              |   |  |   |  |  |   |      |
              =  <>  <  <=  >  >=  IN  simple expr
              |   |  |   |  |  |   |      |
              |   |  |   |  |  |   |      |
              |   v  v   v  v  v   v      |
              ----------------------------->|
```

simple expr

```
  |--> + -->|
  |         |
--------------->term----------------->
  |         |        |   |  |   |
  |--> - -->|        |   |  |   |
                   term  +  -  OR
                     |   |  |   |
                     |   |  |   |
                     |   v  v   v
                     |<----------
```

term

```
-->factor------------------------------->
            |     | |   |    |   |
            |     | |   |    |   |

          factor  * /  DIV  MOD  8

            |     | |   |    |   |
            |     | |   |    |   |
            |     v v   v    v   v
            |<-----------------------
```

factor

```
---------->constant----------->
    |                     |
    |----->variable------>|
    |                     |
    |--->routine call---->|
    |                     |
    |--->(-->expr-->)---->|
    |                     |
    |-->NOT---->factor--->|
    |                     |
    |-->set constructor-->|
```

9.1. Type Compatibility

An operation can only be performed on two operands if their data types are compatible. They are compatible if one of the following conditions is satisfied after coercion:

1) Both types are defined by the same type definition or variable declaration (Sections 7, 8).
2) Both types are subranges of a single enumeration type (Section 7.1).
3) Both types are strings of the same length (Section 7.3).
4) Both types are sets of compatible base types. The empty set is compatible with any set (Section 7.5).
5) A set constant is compatible with any set of the same length.

9.2. Coercion

Coercion is the process of forcing real, integer or set operands of an operator to be of compatible types. Within arithmetic expressions

1. all variables' values are coerced (expanded) to the length of the longest value in the expression,
2. all integer constants are coerced to the length of the longest value in the expression, and
3. all real constants are coerced (truncated or expanded) to be the length of the longest variable's value in the expression.

When parameters are passed

1. if the formal parameter is of type VAR, then on coercion takes place,
2. if the formal parameter is of type constant, then shorter actual parameters are coerced to the length of the formal parameter, and
3. if the formal parameter is of type short real constant, then long real actual parameters are coerced (truncated) to the length of the formal parameter.

10. Statements

Statements define operations on constants and variables:

```
statement

--------------------------------->        SECTION
   |                         |
   |-->compound statement-->|              5
   |                         |
   |---->case statement---->|              7.1
   |                         |
   |---->for statement----->|              7.1
   |                         |
   |----->if statement----->|              7.1.2
   |                         |
   |--->while statement---->|              7.1.2
   |                         |
   |--->repeat statement--->|              7.1.2
   |                         |
   |---->with statement---->|              7.4
   |                         |
   |------>assignment------>|              8
   |                         |
   |---->procedure call---->|              11
```

Empty statements, assignments, and routine calls cannot be divided into smaller statements. They are simple statements. All other statements are structured statements formed by combinations of statements.

An empty statement has no effect.

11. ROUTINES

11. Routines

A *routine* defines a set of parameters and a block that operates on them. In the case of prefix routines (Section 13) and forward declarations, the block is omitted.

routines

```
  |<--- ; <--- procedure <--|
  |                         |
-------------------------------->
  |                         |
  |<--- ; <--- function  <--|
```

There are two kinds of routines, procedures and functions. A *procedure* consists of a procedure heading and a block to be executed when the procedure is called:

procedure

```
                        |-->block----|
                        |            |
--->procedure heading---|            |--->
                        |            |
                        |-->FORWARD--|
```

A *function* consists of a function heading and a block to be executed when the function is called:

function

```
                        |-->block----|
                        |            |
--->function heading----|            |--->
                        |            |
                        |-->FORWARD--|
```

If a routine is referenced before its block is defined, it must be introduced first by means of its heading followed by the symbol FORWARD. The routine can then be completed later by repeating its heading (without the parameter list) followed by the block.

A procedure heading defines the procedure identifier and its parameter list.

procedure heading

```
                              |----------------->|
                              |                  |
-->PROCEDURE-->identifier---->parameter list---->;-->
```

A function heading gives the function identifier, its parameter list, and the function type.

function heading

```
                        |------------------------------->|
                        |                                |
->FUNCTION->identifier-->parameter list->:->identifier-->;->
```

A function computes a value. The value e of a function f is defined by an assignment

f: = e

within the function block.

The function and its value must be of compatible enumeration or pointer types.

parameter list

```
 |----------------------------------------------------------->|
 |                                                            |
 |   |----->|                    |----->|                     |
 |   |      |                    |      |                     |
-->(--->VAR--->identifiers->:--->UNIV--->identifier--->)--->
    |                                                  |
    |<------------------ ; <---------------------------|
```

A parameter list defines the type of parameters on which a routine can operate. Each parameter is specified by its parameter and type identifiers (separated by a colon).

A *variable parameter* represents a variable to which the routine may assign a value. It is prefixed with the word VAR. The parameter declaration

var v1, v2, ... , vn: T

is equivalent to

var v1: T; var v2, ... , vn: T

A *constant parameter* represents an expression that is evaluated when the routine is called. Its value cannot be changed by the routine. A constant parameter is not prefixed with the word VAR.

The parameter declaration

v1, v2, ... , vn: T

is equivalent to

v1: T; v2, ... , vn: T

A parameter is of *universal type* if its type identifier is prefixed with the word UNIV. The meaning of universal types is explained later.

The parameters and variables declared within a routine exist only while it is being executed. They are *temporary* variables.

Function parameters must be constant.

Universal types must be nonlist types.

11.1. Universal Parameters

The prefix UNIV suppresses compatibility checking of parameter and argument types in routine calls (Sections 9, 11).

An argument of type T1 is compatible with a parameter of universal type T2 if both types are nonlist types and represented by the same number of store locations.

The type checking is only suppressed in routine calls. Inside the given routine the parameter is considered to be of non-universal type T2, and outside the routine call the argument is considered to be of non-universal type T1.

routine call

--->identifier--->arguments--->

A routine call specifies the execution of a routine with a set of arguments. It can either be a function call or a procedure call.

A routine call used as a factor in an expression must be a function call. A routine call used as a statement must be a procedure call (Sections 9, 10).

arguments

```
  |------------------------------------>|
  |                                     |
  |           |-->variable-->|          |
  |           |              |          |
-----> ( ----->|              |-----> ) ----->
         |    |              |  |
         |    |---->expr---->|  |
         |                      |
         |<------- , <--------|
```

An argument list defines the arguments used in a routine call. The number of arguments must equal the number of parameters specified in the routine. The arguments are substituted for the parameters before the routine is executed.

Arguments corresponding to variable and constant parameters must be variables and expressions, respectively. The selection of variable arguments and the evaluation of constant arguments are done once only (before the routine is executed).

The argument types must be compatible with the corresponding parameter types with the following exceptions:

1. An argument corresponding to a constant string parameter may be a string of any length.
2. An argument corresponding to a universal parameter may be of any nonlist type that occupies the same number of store locations as the parameter type.

12. Scope Rules

A *scope* is a region of program text in which an identifier is used with a single meaning. An identifier must be *introduced* before it is *used*. (The only exception to this rule is a pointer type: it may refer to a type that has not yet been defined.)

A scope is either a program, a routine or a with statement. A *program* or *routine* introduces identifiers by *declaration*; a *with statement* does it by *selection* (Sections 5, 7.4, 7.6, 11).

When a scope is defined within another scope we have an *outer scope* and an *inner scope* that are *nested*. An identifier can only be introduced with one meaning in a scope. It can, however, be introduced with another meaning in an inner scope. In that case, the inner meaning applies in the inner scope and the outer meaning applies in the outer scope.

Routines cannot be nested. Within a routine, with statements can be nested. This leads to the following *hierarchy of scopes*:

```
(program
  (non-nested routines
    (nested with statements)))
```

A *program* can use

(1) any standard identifier.

(2) constant, type, and routine identifiers introduced within and after the prefix (Section 13).

A routine can use

(1), (2) defined above and

(3) all identifiers introduced within the routine itself.

A with statement can use

(1), (2), (3) defined above and

(4) all identifiers introduced by the with statement itself and by its outer with statements.

The phrase "all identifiers introduced in its outer scopes" should be qualified with the phrase "unless these identifiers are used with different meanings in these scopes. In that case, the innermost meaning of each identifier applies in the given scope."

13. Sequential Programs

A sequential program consists of a prefix followed by a block:

```
program

---->prefix-->block--> . -->
```

The prefix defines the program's interface to the operating system. This interface consists of constant, type, and routine definitions:

```
prefix

 |<-const definitions<-|
 |                     |
------------------------>prefix routines->program heading->
 |                     |
 |<-type definitions<--|
```

```
prefix routines

-->|<--procedure heading<-----> escape heading-->
   |                       |
   |<--function heading<---|
   |                       |
   |---------------------->|
```

Prefix routines consist only of procedure or function headings. The prefix routine blocks are defined within the operating system. They can be called as other routines by the program (Section 11).

The program heading gives the program identifier and its parameter list:

<u>program heading</u>

```
-->PROGRAM-->identifier-->parameter list--> ; -->
```

<u>escape heading</u>

```
->ESCAPE-->id-->(-->escape list-->;-->id-->:-->INTEGER-->)->
```

<u>escape list</u>

```
   |-------->|                       |--------->|
   |         |                       |          |
----->VAR----->id list-----> : ----->UNIV----->id----->
  |                                                   |
  |<---------------------- ; <------------------------|
```

APPENDIX A

ASCII CHARACTER SET

<u>ASCII CHARACTER SET</u>

0	nul	32		64	@	96	
1	sch	33	|	65	A	97	a
2	stx	34	"	66	B	98	b
3	etx	35	#	67	C	99	c
4	eot	36	$	68	D	100	d
5	enq	37	%	69	E	101	e
6	ack	38	&	70	F	102	f
7	bel	39	'	71	G	103	g
8	bs	40	(	72	H	104	h
9	ht	41	)	73	I	105	i
10	lf	42	*	74	J	106	j
11	vt	43	+	75	K	107	k
12	ff	44	,	76	L	108	l
13	cr	45	-	77	M	109	m
14	so	46	.	78	N	110	n
15	si	47	/	79	O	111	o
16	dle	48	0	80	P	112	p
17	dc1	49	1	81	Q	113	q
18	dc2	50	2	82	R	114	r
19	dc3	51	3	83	S	115	s
20	dc4	52	4	84	T	116	t
21	nak	53	5	85	U	117	u
22	syn	54	6	86	V	118	v
23	etb	55	7	87	W	119	w
24	can	56	8	88	X	120	x
25	em	57	9	89	Y	121	y
26	sub	58	:	90	Z	122	z
27	esc	59	;	91	[	123	
28	fs	60	<	92		124	
29	gs	61	=	93	]	125	
30	rs	62	>	94	¬	126	
31	us	63	?	95	_	127	del

INDEX

APPENDIX B:

PASCAL/S SYNTAX GRAPHS

SYNTAX GRAPHS

SOURCE

1. Program

```
----> prefix ----> block ----> . ---->
```

2. prefix

```
   |<--   const    <--|
   |   declarations   |
   |                  |        prefix         program
---------------------------> routines --> declaration --->
   |                  |
   |      type        |
   |<--declarations<--|
```

3. prefix routines

```
   |<--procedure declarations<--|
   |                            |
----------------------------------->escape declarations--->
   |                            |
   |<--function declarations<---|
```

4. block

```
-----> declarations -----> body
```

5. declarations

```
   |<--  constant  <--|
   |   declarations   | |-------------->| |-------------->|
   |                  | |               | |               |
-----------------------------> variable -----> routine   --->
   |                  |   declarations      declarations
   |       type       |
   |<--declarations<--|
```

6. constant declarations

```
---->CONST---->id---> = --->constant---> ; ----->
          |                                  |
          |<---------------------------------|
```

7. type declarations

```
---->TYPE---->id---> = --->type---> ; ----->
         |                              |
         |<-----------------------------|
```

8. type

```
---------------> id ---------------->
   |                            |
   |---> enumeration type ---->|
   |                            |
   |-----> subrange type ----->|
   |                            |
   |------> set type --------->|
   |                            |
   |-----> array type -------->|
   |                            |
   |-----> record type ------->|
   |                            |
   |------> real type -------->|
   |                            |
   |-----> sreal type -------->|
   |                            |
   |------> ptr type --------->|
```

9. enumeration type

```
-------> ( ---> id list ---> ) ------->
   |                               |
   |-------> CHAR --------------->|
   |                               |
   |-------> BOOLEAN ------------>|
   |                               |
   |-------> INTEGER ------------>|
```

10. subrange type

```
-----> constant -----> .. -----> constant ----->
```

11. set type

```
-----> SET OF -----> type ----->
```

12. <u>array type</u>

```
-->ARRAY---> (. ----->enumeration type-----> .) -->OF-->type
         |        | |                    | |       |
         |        | |-->subrange type-->| |       |
         |        |                        |       |
         |        |<--------- , <---------|       |
         |                                         |
         |                                         |
         |-> [ ----->enumeration type-----> ] ->|
                  | |                    | |
                  | |-->subrange type-->| |
                  |                        |
                  |<--------- , <---------|
```

13. <u>record type</u>

```
----> RECORD ----> field list ----> END
```

14. <u>field list</u>

```
  |---------------->|
  |                 |
-----> fixed part -----> variant part ----->
                    |                     |
                    |------------------->|
```

15. <u>fixed part</u>

```
----->id list-----> : ----->type----->
  |                                |
  |<-------------- ; <-----------|
```

16. variant part

```
--->CASE--->id---> : --->id--->OF--->variant--->
                                    |           |
                                    |<--- ; <--|
```

17. variant

```
--->labels---> : ---> ( --->field list---> ) --->
```

18. labels

```
-----> enumeration constant ----->
  |                          |
  |<---------- ; <----------|
```

19. pointer type

```
-----> @ ---------> type ----->
  |              |
  |--> ¬ --->|
```

20. variable declaration

```
--->VAR--->id list---> : --->type---> ; --->
        |                              |
        |<-----------------------------|
```

21. id list

```
-----> id ------>
  |          |
  |<--- , <--|
```

22. routine declarations

```
   |<--- ; <--- function declarations <----|
   |                                       |
------------------------------------------------>
   |                                       |
   |<--- ; <--- procedure declarations <---|
```

23. procedure declaration

```
-->PROCEDURE-->id-->parm list--> ; ------->block----------->
                   |            |     |                  |
                   |----------->|     |-->FORWARD--> ; -->|
```

24. function declaration

```
              |----------------->|
              |                  |
->FUNCTION->id->parm list-> : ->id-> ; ------>block-------->
                                         |                |
                                         |->FORWARD-> ; ->|
```

25. program declaration

```
---> PROGRAM ---> id ---> parm list ---> ; --->
```

26. <u>parm list</u>

```
         |----->|                 |------->|
         |      |                 |        |
--> ( ----->VAR--->id list--> : --->UNIV---->id---> ) -->
       |                                       |
       |<------ ; <----------------------------|
```

27. <u>escape declaration</u>

```
-->ESCAPE-->id--> ( -->escape list--> ; -->id--|
                                               |
                                               |
        |<-------------------------------------|
        |
        |-->integer--> ) --> ; ----->
```

28. <u>escape list</u>

```
   |----->|                  |------>|
   |      |                  |       |
----->VAR--->id list---> : --->UNIV--->id----->
 |                                          |
 |<----------------- ; <-----------------|
```

29. <u>body</u>

```
----> BEGIN ----> stat list ----> END
```

30. <u>stat list</u>

```
-----> stat ----->
  |          |
  |<--- ; <---|
```

31. stat

```
--------------------------------->
   |                       |
   |--> assignment ------->|
   |                       |
   |--> compound stat ---->|
   |                       |
   |--> if stat ---------->|
   |                       |
   |--> case stat -------->|
   |                       |
   |--> while stat ------->|
   |                       |
   |--> repeat stat ------>|
   |                       |
   |--> for stat --------->|
   |                       |
   |--> with stat -------->|
   |                       |
   |--> procedure call --->|
```

32. assignment

```
-----> variable -----> := -----> expr ----->
```

33. procedure call

```
-----> id -----> arg list ----->
```

34. arg list

```
  |------------------------->|
  |                          |
-----> ( ----->expr-----> ) ----->
           |          |
           |<-- , <---|
```

35. compound stat

```
-----> BEGIN -----> stat list -----> END
```

36. if stat

```
                               |-------------->|
                               |               |
--->IF--->expr--->THEN--->stat--->ELSE--->stat----->
```

37. case stat

```
--->CASE-->expr-->OF----->constant---> : --->stat--->END--->
                     | |             |               |
                     | |<--- , <---|                 |
                     |                               |
                     |<--------------- ; <--------|
```

38. while stat

```
----> WHILE ----> expr ----> DO ----> stat
```

39. repeat stat

```
----> REPEAT ----> stat list ----> UNTIL ----> expr ---->
```

40. for stat

```
                                   |---->TO--->|
->FOR-->id--> := -->expr--|             |->expr-->DO-->stat-->
                                   |->DOWN TO->|
```

41. with stat

```
--->WITH--->variable--->DO--->stat--->
         |           |
         |<--- , <---|
```

42. expr

```
--->sexpr-------------------------------------->
            |   |   |   |   |   |   |    |
            |   |   |   |   |   |   |    |
            =   <>  <   <=  >   >=  IN  sexpr
            |   |   |   |   |   |   |    |
            |   |   |   |   |   |   |    |
            V   V   V   V   V   V   V    |
            ---------------------------->|
```

43. sexpr

```
    |--> + -->|
    |         |
------------------>term---------------------->
    |         |         |    |    |    |
    |--> - -->|         |    |    |    |
                      term   +    -    OR
                        |    |    |    |
                        |    |    |    |
                        |    V    V    V
                        |<------------
```

44. term

```
--->factor---------------------------------->
              |      |  |   |     |   |
              |      |  |   |     |   |

           factor    *  /  DIV   MOD  &

              |      |  |   |     |   |
              |      |  |   |     |   |
              |      V  V   V     V   V
              |<--------------------------
```

45. factor

```
--------------> constant ----------------->
    |                                  |
    |---------> variable ----------->|
    |                                  |
    |------> function call --------->|
    |                                  |
    |-----> ( ---> expr ---> ) ----->|
    |                                  |
    |------> NUT -----> factor ----->|
    |                                  |
    |--> (. -----> expr -----> .) -->|
    |         | |           | |        |
    |         | |<---  ,  ---| |        |
    |         |               |        |
    |         |--------------->|        |
    |                                  |
    |                                  |
    |---> [ -----> expr -----> ] --->|
              | |           | |
              | |<---  ,  ---| |
              |               |
              |--------------->|
```

46. function call

```
-----> id -----> arg list ----->
```

47. variatle

```
---> id ------------------------------------->
         |                                |
         |<---- id <--- . <----------------|
         |                                |
         |<-- .) <--- expr <--- (. <--|
         |              |         |           |
         |              |--> , -->|           |
         |                                |
         |                                |
         |              |--- @ <---|           |
         |<---------|          |<------|
                        |--- ¬ <---|
```

48. enumeration constant

```
---------------> id ----------------->
     |                              |
     |-----> char constant ------>|
     |                              |
     |----> boolean constant ---->|
     |                              |
     |----> integer constant ---->|
```

49. constant

```
    |-----------> id ----------->|
    |                              |
----|---------> string --------->|---->
    |                              |
    |----> scalar constant ----->|
    |                              |
    |------> set constant ------>|
```

50. id

```
-----> letter --------------------------->
                 |                |
                 |<--- letter <---|
                 |                |
                 |<--- digit <----|
```

51. string

```
             |---------->character---------->|
-----> ' ---|                                 |----> ' ----->
           | |---> (: --->integer---> :) --->|  |
           |                                    |
           |<-----------------------------------|
```

52. scalar constant

```
       |-----> real constant ----->|
----->|                            |----->
       |-----> index constant ---->|
```

53. real constant

```
             |----------------->|
             |                  |   |-> + -->|
   digit     |       digit      |   |        |   digit
-->sequence---> . -->sequence--->E--------------->sequence-->
                             |      |        |             |
                             |      |-> - -->|             |
                             |                             |
                             |---------------------------->|
```

54. digit sequence

```
-----> digit ----->
  |            |
  |<-----------|
```

55. index constant

```
-------------> integer --------------->
    |                             |
    |-----> char constant ------->|
    |                             |
    |----> boolean constant ----->|
```

56. boolean constant

```
       |---> TRUE --->|
----->|              |----->
       |--> FALSE --->|
```

57. integer

```
  |--> + -->|
  |         |
---------------->digit sequence----->
  |         |
  |--> - -->|
```

58. char ccnstant

```
          |--------->character---------->|
---> ' ---|                               |-----> ' ----->
          |--> (: --->integer---> :) --->|
```

59. <u>set constant</u>

```
-----> [ --->B--> ' -->bits--> ' ---> ] ------->
  |                                          |
  |--> 8. -->B--> ' -->bits--> ' --> .) --->|
```

60. <u>bits</u>

```
----------> bits ---------------------->
  | |                               | |
  | |-->repetition factor--->bit--->| |
  |                                   |
  |<----------------------------------|
```

61. <u>repetition factor</u>

```
----> ( ----> integer ----> ) ---->
```

62. <u>bit</u>

```
---------> 0 --------->
    |               |
    |----> 1 ---->|
```

63. <u>separator</u>

```
----------------> space ----------------->
    |                                  |
    |----------> end of line ---------->|
    |                                  |
    |----> " ----> comment ----> " ---->|
    |                                  |
    |----> (* ---> comment ---> *) ---->|
    |                                  |
    |-->    ------> comment ---->   ---->|
```

OUTPUT PASS 1

1. program

```
----> prefix ----> block ----> PERIOD ----> EOM ---->
```

2. prefix

```
   |<--   const    <--|
   |   declarations   |
   |                  |   prefix      program
------------------------->routines-->declaration--->
   |                  |
   |       type       |
   |<--declarations<--|
```

3. prefix routines

```
   |<--procedure declarations<--|
   |                            |
----------------------------------->escape declarations--->
   |                            |
   |<--function declarations<---|
```

4. block

```
----> declarations ----> body ---->
```

5. <u>declarations</u>

```
   |<--  constant  <--|
   |   declarations   | |-------------->| |------------->|
   |                  | |               | |              |
----------------------------> variable ----->  routine  --->
   |                  |     declarations     declarations
   |                  |
   |       type       |
   |<--declarations<--|
```

6. <u>constant declarations</u>

```
--->CONST--->ID(spix)--->EQ--->constant--->SEMICOLON--->
          |                                          |
          |<-----------------------------------------|
```

7. <u>type declarations</u>

```
--->TYPE--->ID(spix)--->EQ--->type--->SEMICOLON--->
         |                                    |
         |<-----------------------------------|
```

8. type

```
-----------> ID(spix) ------------->
   |                              |
   |---> enumeration type --->|
   |                              |
   |----> subrange type ----->|
   |                              |
   |------> set type -------->|
   |                              |
   |-----> array type ------->|
   |                              |
   |-----> record type ------>|
   |                              |
   |------> real type ------->|
   |                              |
   |-----> sreal type ------->|
   |                              |
   |------> ptr type -------->|
```

9. enumeration type

```
-----> OPEN ---> id list ---> CLOSE ----->
  |                                    |
  |--------------> CHAR -------------->|
  |                                    |
  |-------------> BOOLEAN ----------->|
  |                                    |
  |-------------> INTEGER ----------->|
```

10. subrange type

```
----> constant ----> UPTO ----> constant ---->
```

11. set type

```
----> SET ----> OF ----> type ---->
```

12. array type

```
--->ARRAY--->SUB----->enumeration type----->BUS-->OF--->TYPE
                 | |                     | |
                 | |-->subrange type-->| |
                 |                         |
                 |<------COMMA<----------|
```

13. record type

```
----> RECORD ----> field list ----> END
```

14. field list

```
  |--------------->|
  |                |
----->fixed part----->variant part----->
                 |                |
                 |---------------->|
```

15. fixed part

```
----->id list----->COLON----->type----->
  |                                |
  |<-----------SEMICOLON<-----------|
```

16. variant part

```
-->CASE-->ID(spix)-->COLON-->ID(spix)-->OF---->variant---->
                                           |              |
                                           |<-SEMICOLON-|
```

17. variant

```
--> labels --> COLON --> OPEN --> field list --> CLOSE -->
```

18. labels

```
-----> enumeration constant ----->
  |                             |
  |<--------- COMMA <---------|
```

19. pointer type

```
----> ARROW ----> type
```

20. variable declarations

```
--> VAR ---> id list --> COLON --> type --> SEMICOLON --->
         |                                               |
         |<----------------------------------------------|
```

21. id list

```
-----> ID(spix) ----->
  |                |
  |<--- COMMA <---|
```

22. routine declaration

```
  |<--SEMICOLCN<--procedure declaration<--|
  |                                       |
------------------------------------------------>
  |                                       |
  |<--SEMICOLCN<---function declaration<--|
```

23. procedure declaration

```
---->PROCEDURE---->ID (spix)---->parm---->SEMICOLON---|
                             |   list  |              |
                             |         |              |
                             |-------->|              |
                                                      |
         |<-------------------------------------------|
         |
         |------------>block-------------------->
             |                           |
             |-->FORWARD-->SEMICOLON--->|
```

24. function declaration

```
                          |---------------------------->|
                          |                             |
--->FUNCTION--->ID (spix)--->parm--->COLON--->ID (spix)----|
                             list                          |
                                                           |
      |<---------------------------------------------------|
      |
      |--->SEMICOLON------->block---------------------------->
                      |                               |
                      |----->FORWARD--->SEMICOLON--->|
```

25. program declaration

```
--> PROGRAM --> ID(spix) --> parm list --> SEMICOLON --->
```

26. <u>parm list</u>

```
              |----->|                      |------>|
              |      |                      |       |
-->OPEN---->VAR-->id list-->COLON-->UNIV-->ID(spix)-->CLOSE
           |                                           |
           |<---------------SEMICOLON<-----------------|
```

27. <u>escape declaration</u>

```
-->ESCAPE-->ID(spix)-->OPEN-->escape list-->SEMICOLON---|
                                                        |
       |<-----------------------------------------------|
       |
       |-->ID(integer spix)-->CLOSE-->SEMICOLON------------>
```

28. <u>escape list</u>

```
   |------>|                    |------->|
   |       |                    |        |
----->VAR----->id list--->COLON--->UNIV---->id---->
 |                                      |
 |<-----------------SEMICOLON<----------------|
```

29. <u>body</u>

```
----> BEGIN ----> stat list ----> END ---->
```

30. <u>stat list</u>

```
--------->stat--------->
   |            |
   |<---SEMICOLON<--|
```

31. stat

```
------------------------------------>
   |                              |
   |-----> assignment ----->|
   |                              |
   |---> procedure call --->|
   |                              |
   |---> compound stat ---->|
   |                              |
   |-------> if stat ------>|
   |                              |
   |------> case stat ----->|
   |                              |
   |------> while stat ---->|
   |                              |
   |-----> repeat stat ---->|
   |                              |
   |------> for stat ------>|
   |                              |
   |------> with stat ----->|
```

32. assignment

```
---> variatle ---> BECOMES ---> expr --->
```

33. procedure call

```
---> ID(spix) ---> arg list --->
```

34. arg list

```
  |------------------------------->|
  |                                |
----->OPEN----->expr------>CLOSE----->
              |           |
              |<--COMMA<--|
```

35. compound statement

```
---> BEGIN ---> stat list ---> END --->
```

36. if stat

```
                               |--------------->|
                               |                |
--->IF--->expr--->THEN--->stat--->ELSE--->stat---->
```

37. case stat

```
-->CASE-->expr-->OF----->constant--->COLON-->stat--->END-->
                    | |           |               |
                    | |<--COMMA---|               |
                    |                             |
                    |<---------SEMICOLON<---------|
```

38. while stat

```
---> WHILE ---> expr ---> DO ---> stat --->
```

39. repeat stat

```
---> REPEAT ---> stat list ---> UNTIL ---> expr --->
```

40. for stat

```
->FOR->ID(spix)->BECOMES->expr--->DOWNTO--->expr->DO->stat->
                                  |         |
                                  |-->TO--->|
```

41. with stat

```
----->WITH----->variable----->DO----->stat----->
            |             |
            |<---COMMA<---|
```

42. expr

```
             |-------------------->|
             |                     |
----->sexpr---->expr op---->sexpr----->
```

exprop: EQ NE LE GE LT GT IN

43. sexpr

```
  |-->UPLUS--->|          |----------------------->|
  |            |          |                        |
------------------>term----->sexpr op--->term----->
  |            |            |                   |
  |-->UMINUS-->|            |<------------------|
```

sexpr op: PLUS MINUS OR

44. term

```
           |----------------------->|
           |                        |
--->factor----->term op--->factor----->
             |                    |
             |<-------------------|
```

term cp: STAR SLASH DIV MOD AND

45. factor

```
---------------> constant ------------------>
   |                                   |
   |-----------> variable ----------->|
                                       |
   |--------> function call --------->|
   |                                   |
   |---> OPEN --> expr --> CLOSE ---->|
   |                                   |
   |--------> NOT ---> factor ------->|
   |                                   |
   |                                   |
   |          |------------->|         |
   |          |              |         |
   |--> SUB -----> expr -----> BUS -->|
                |         |
                |<-COMMA--|
```

46. function call

```
----> ID(spix) ----> arg list ---->
```

47. variable

```
--->ID(spix)---------------------------------------->
                 |                                |
                 |<----ID(spix)<-----PERIOD<-----|
                 |                                |
                 |<---BUS<-----expr<-----SUB<----|
                 |          |          |          |
                 |          |<--COMMA<--|         |
                 |                                |
                 |                                |
                 |<------------ARROW<------------|
```

48. enumeration constant

```
-----------> ID(spix) ---------------->
    |                             |
    |---> character constant --->|
    |                             |
    |----> boolean constant ---->|
    |                             |
    |----> integer constant ---->|
```

49. constant

```
----------> ID(spix) ------------->
   |                            |
   |----> STRING(length) ----->|
   |                            |
   |----> scalar constant ---->|
   |                            |
   |-----> set constant ------>|
```

52. scalar constant

```
----->REAL----->LARGE CONSTANT(value)----->
  |                                      |
  |---------->index constant---------->|
```

55. <u>index constant</u>

```
----->INTEGER--->LARGE CONSTANT--->(value)----->
  |                                           |
  |--->CHAR--->(value)--------------------->|
```

59. <u>set constant</u>

```
---> SUB ---> SET CONST(value) ---> BUS
```

OUTPUT PASS 2

1. program

```
----> prefix ----> block ----> EOM ---->
```

2. prefix

```
   |<--   const    <--|
   |   declarations   |
   |                  |   prefix      program
------------------------->routines-->declaration--->
   |                  |
   |        type      |
   |<--declarations<--|
```

3. prefix routines

```
   |<--procedure declarations<--|
   |                            |
------------------------------------->escape declarations--->
   |                            |
   |<--function declarations<---|
```

4. block

```
-----> declarations -----> body
```

5. declarations

```
    |<--  constant  <--|
    |   declarations   | |------------->| |-------------->|
    |                  | |              | |               |
--------------------------  variable  ----->  routine  --->
    |                  | declarations      declarations
    |       type       |
    |<--declarations<--|
```

6. constant declarations

```
----->CONST ID(spix)--->constant--->CONST DEF----->
  |                                            |
  |<-------------------------------------------|
```

7. type declarations

```
----->TYPE ID(spix)--->type--->TYPE DEF----->
  |                                      |
  |<-------------------------------------|
```

8. type

```
----------> TYPE(spix) ------------>
    |                              |
    |---> enumeration type --->|
    |                              |
    |----> subrange type ----->|
    |                              |
    |------> set type -------->|
    |                              |
    |-----> array type ------->|
    |                              |
    |-----> record type ------>|
    |                              |
    |------> real type ------->|
    |                              |
    |-----> sreal type ------->|
    |                              |
    |------> ptr type -------->|
```

9. enumeration type

```
----->ENUM----->ENUM ID(spix)----->ENUM DEF----->
               |                  |
               |<-----------------|
```

10. subrange type

```
----> constant ----> constant ----> SUBR DEF ---->
```

11. set type

```
-----> type -----> SET DEF ----->
```

12. <u>array type</u>

```
---> type ---> type ---> ARRAY DEF --->
```

13. <u>record type</u>

```
---> REC ---> field list ---> REC DEF --->
```

14. <u>field list</u>

```
  |---------------->|
  |                 |
----->fixed part----->variant part----->
                 |                   |
                 |------------------>|
```

15. <u>fixed part</u>

```
--->FIELD ID(spix)--->type--->FIELD LIST(number)--->
 |                |                               |
 |<---------------|                               |
 |                                                |
 |<-----------------------------------------------|
```

16. <u>variant part</u>

```
-->TAG ID(spix)-->TAG TYPE(spix)-->TAG DEF-->variant-->
```

17. variant

```
-->VARNT--->enumeration-->LABEL--->LABEL END---|
        |    constant          |               |
        |                      |               |
        |<---------------------|               |
                                               |
                      |<-----------------------|
                      |
                      |-->field list-->VARNT END---->
```

19. pointer type

```
-----> POINTER(spix) ----->
```

20. variable declaration

```
--->VAR ID(spix)--->type--->VAR LIST(number)--->
 |              |                            |
 |<-------------|                            |
 |                                           |
 |<------------------------------------------|
```

22. routine declarations

```
    |<---procedure declarations<---|
    |                              |
--------------------------------------->
    |                              |
    |<----function declarations<---|
```

23. procedure declaration

```
-->PROC ID(spix)->parm list-->PROC DEF-->block-->PROC END-->
```

24. <u>function declaration</u>

```
-->FUNC ID(spix)-->parm list-->FUNC DEF(spix)-->block--|
                                                       |
                |<-------------------------------------|
                |
                |--->FUNC END----->
```

25. <u>program declaration</u>

```
---> PROG ID(spix) ---> parm list ---> PROG DEF --->
```

26. <u>parm list</u>

```
--->PARM ID(spix)--->PARM/UNIV TYPE(spix)---|
                                            |
         |<---------------------------------|
         |
         |---->V/C PARM LIST(length)---------->
```

27. <u>escape declaration</u>

```
--->ESCAPE ID(spix)--->parm list--->PARM ID(spix)---|
                                                    |
    |<----------------------------------------------|
    |
    |-->PARM TYPE(integer spix)-->C/V PARM LIST(number)-->
```

29. <u>body</u>

```
---> BODY ---> stat ---> BODY END --->
```

31. stat

```
----------------------------------->
   |                              |
   |--> assignment ------>|
   |                              |
   |--> if stat --------->|
   |                              |
   |--> case stat ------->|
   |                              |
   |--> while stat ------>|
   |                              |
   |--> repeat stat ----->|
   |                              |
   |--> for stat -------->|
   |                              |
   |--> with stat ------->|
   |                              |
   |--> procedure call -->|
```

32. assignment

```
---> name ---> ANAME ---> expr ---> STORE --->
```

33. procedure call

```
--> name --> CALL NAME --> arg list --> CALL -->
```

34. arg list

```
  |----------------------------------->|
  |                                    |
----->ARG LIST----->expr----->ARG----->
                  |                 |
                  |<----------------|
```

36. <u>if</u> <u>stat</u>

```
--->expr--->FALSE JUMP(L1)--->stat---|
                                     |
  |<---------------------------------|
  |
  |
  |   |-->DEF LABEL(L1)----------------------------->|
  |---|                                              |---->
      |-->JUMP DEF(L2,L1)--->stat--->DEF LABEL(L2)-->|
```

37. <u>case</u> <u>stat</u>

```
                       |<-----------------------------------|
                       |                                    |
                       |             |<----------------|    |
                       |             |                 |    |
-->expr->CASE JUMP(L0)-->DEF CASE(L1)-->constant->CASE---   |
                                                        |   |
                                                        |   |
          |<--------------------------------------------|   |
          |                                                 |
          |-->stat-->JUMP(Ln)-->END CASE(L0,Ln)--->         |
                            |                               |
                            |------------------------------>|
```

38. <u>while</u> <u>stat</u>

```
--->DEF LABEL(L1)--->expr--->FALSE JUMP(L2)---|
                                              |
      |<--------------------------------------|
      |
      |-->stat--->JUMP DEF(L1,L2)--------->
```

39. <u>repeat</u> <u>stat</u>

```
--> DEF LABEL(L) --> stat --> expr --> FALSE JUMP(L) -->
```

40. <u>for stat</u>

```
--->name-->ADDRESS-->expr-->FOR STORE-->expr----|
                                                |
  |<--------------------------------------------|
  |
  |->FOR LIM(L1,COMP,L2)-->stat-->FOR UP/DOWN(L1,L2)--->
```

41. <u>with stat</u>

```
--> WITH VAR --> name --> WITH TEMP --> stat --> WITH -->
```

42. <u>expr</u>

```
          |---------------------------->|
          |                             |
--->sexpr--->VALUE--->sexpr--->expr op--->
```

43. <u>sexpr</u>

```
                       |------------------------------>|
         |->UPLUS--|   |                               |
-->term--|         |---->VALUE--->term--->sexpr op----->
         |->UMINUS-|            |                    |
                                |<-------------------|
```

44. <u>term</u>

```
           |-------------------------------->|
           |                                 |
--->factor--->VALUE--->factor--->term op----->
                     |                    |
                     |<-------------------|
```

45. factor

```
----------------------->name-->FNAME----------------------->
    |                                                   |
    |----------------->factor constant--------------->|
    |                                                   |
    |------------------>function call---------------->|
    |                                                   |
    |---------------------->expr--------------------->|
    |                                                   |
    |------------------>factor-->NOT----------------->|
    |                                                   |
    |-->EMPTY SET-->expr-->INCLUDE-->END BUILDSET--->|
```

factor constant: constant w/ 'F' prefixed to all
                  terminal symbols.

46. function call

```
--> name --> FUNCTION --> arg list --> CALL FUNC -->
```

47. name

```
--->NAME(spix)------------------------------------->
                  |                          |
                  |<--------COMP(spix)<-------|
                  |                          |
                  |<--SUB<--expr<--address<---|
                  |                          |
                  |<-------ARROW(spix)<-------|
```

48. enumeration constant

```
----> CONST ----> (spix,sign) ---->
```

49. <u>constant</u>

```
------->CONSTANT(spix,sign)-------->
    |                         |
    |----->STRING(length)----->|
    |                         |
    |---->scalar constant----->|
    |                         |
    |---->SET CONST(length)--->|
```

52. <u>scalar</u> <u>constant</u>

```
     |-->REAL--->LCONST(value)---|
--->|                            |---->
     |-->index constant----------|
```

55. <u>index</u> <u>constant</u>

```
----->INTEGER(value)----->
  |                   |
  |--->CHAR(value)--->|
```

59. <u>set</u> <u>ccnstant</u>

```
----> SET CCNST(length) ---->
```

OUTPUT PASS 3

1. program

```
----> prefix ----> block ----> EOM
```

2. prefix

```
   |<--    const    <--|
   |   declarations    |
   |                   |   prefix     program
------------------------->routines-->declaration--->
   |                   |
   |       type        |
   |<--declarations<--|
```

3. prefix routines

```
   |<--procedure declarations<--|
   |                            |
------------------------------------>escape declarations--->
   |                            |
   |<--function declarations<---|
```

4. block

```
----> declarations ----> body ---->
```

5. declarations

```
--->    type    -----> variable -----> routines ---->
 |  declarations | ¬  declarations | ¬  declarations  |
 |               | |               | |                |
 |               V |               V |                |
 |------------------------------------------------------>|
```

7. type declarations

```
---->type---->TYPE DEF---->
 |                    |
 |<-------------------|
```

8. type

```
------------------>TYPE(noun)-------------------->
    |                                         |
    |---------->ENUM DEF(noun,max)------------>|
    |                                         |
    |-->SUBR DEF(noun,"range" noun,min,max)--->|
    |                                         |
    |--------->type-->SET DEF(noun)----------->|
    |                                         |
    |----->type-->type-->ARRAY DEF(noun)------>|
    |                                         |
    |------------->record type---------------->|
    |                                         |
    |------------>pointer type---------------->|
```

13. record type

```
---> REC ---> field list ---> REC DEF(noun) --->
```

14. <u>fieldlist</u>

```
  |--------------->|
  |                |
----->fixed part----->variant part----->
                   |               |
                   |---------------->|
```

15. <u>fixed part</u>

```
----->NEW NOUN(noun)----->type--->FIELD LIST(number)----->
 | |                  |                                |
 | |<-----------------|                                |
 |                                                     |
 |<----------------------------------------------------|
```

16. <u>variant part</u>

```
-->NEW NOUN(noun)-->TAG DEF(noun)--->variant--->PART END-->
                                  |          |
                                  |<---------|
```

17. <u>variant</u>

```
-->NEW NOUN(noun)-->type-->FIELD LIST(number)-->VARNT END-->
```

19. <u>pointer type</u>

```
----> POINTER(noun) ---->
```

20. variable declaration

```
----->NEW NOUN(noun)--->type--->VAR LIST(number)----->
 | |                |                             |
 | |<---------------|                             |
 |                                                |
 |<-----------------------------------------------|
```

22. routine declarations

```
    |<---procedure declarations<----|
    |                               |
------------------------------------------>
    |                               |
    |<----function declarations<----|
```

23. procedure declaration

```
--> parm list --> PROC/(f) DEF(noun) --> block -->
```

24. function declaration

```
-->parm list-->FUNC/(f) DEF("type" noun,noun)-->block-->
```

25. program declaration

```
-->parm list-->PROG DEF(noun)--->FWD DEF(noun)--->
                                 |              |
                                 |<-------------|
```

26. parm list

```
----------------------------------------------------------->
  |
  |
  | |<------------------------------------------------|
  | |                                                 |
  |------>NEW NOUN(noun)--->PARM/UNIV TYPE(noun)---   |
       |                 |                        |   |
       |<----------------|                        |   |
                                                  |   |
               |<---------------------------------|   |
               |                                      |
               |--->C/V PARM LIST(number)---->        |
                                           |          |
                                           |--------->|
```

27. escape declaration

```
-->parm list-->NEW NOUN(noun)-->PARM TYPE(integer name)--|
                                                         |
                        |<-------------------------------|
                        |
                        |---->CPARM LIST(number)------>
```

29. body

```
---> BODY ---> stat ---> BODY END --->
```

31. stat

```
------------------------------------>
   |                              |
   |<----- assignment <-----|
   |                              |
   |<------ proc call <-----|
   |                              |
   |<------- if stat <------|
   |                              |
   |<------ case stat <-----|
   |                              |
   |<----- while stat <-----|
   |                              |
   |<---- repeat stat <-----|
   |                              |
   |<------ for stat <------|
   |                              |
   |<------ with stat <-----|
```

32. assignment

```
              |--------->ADDRESS--------->|
--->name--|                               |-->expr-->STORE
              |-->RESULT("type" noun)-->|
```

33. procedure call

```
---> name ---> arg list ---> CALL PROC/PROG --->
```

34. arg list

```
                |------------------------------------------->|
                |                                            |
-->ARG LIST----->expr-->parm("parm" noun,"type" noun)----->
                  |                                        |
                  |<---------------------------------------|
```

36. <u>if stat</u>

```
--->expr--->FALSE JUMP(l1)--->stat---|
                                     |
                                     |
  |<---------------------------------|
  |
  |   |-->DEF LABEL(l1)----------------------------->|
  |---|                                              |---->
      |-->JUMP DEF(l2,l1)--->stat--->DEF LABEL(l2)-->|
```

37. <u>case stat</u>

```
                        |<----------------------------|
                        |                             |
-->expr-->CASE JUMP(l0)--->DEF LABEL(li)--|           |
                                          |           |
                                          |           |
    |<------------------------------------|           |
    |                           ¬                     |
    |                           |                     |
    |--->CHK TYPE("type" noun)--->stat-->JUMP(ln)-----|
                                                    |
                                                    |
    |<----------------------------------------------|
    |
    |--->CASE LIST(l0,min,max,l1,...,ln)------->
```

38. <u>while stat</u>

```
-->DEF LABEL(l1)-->expr-->FALSE JUMP(l2)-->stat---|
                                                  |
                           |<---------------------|
                           |
                           |--->JUMP DEF(l1,l2)-------->
```

39. <u>repeat stat</u>

```
-->DEF LABEL(l)-->stat-->expr-->FALSE JUMP(l)-->
```

40. <u>for stat</u>

```
--->name--->ADDRESS--->expr--->FOR STORE--->expr---|
                                                   |
    |<---------------------------------------------|
    |
    |-->FOR LIM(l1,comp,l2)-->stat-->FOR UP/DOWN(l1,l2)-->
```

41. <u>with stat</u>

```
-->WITH VAR-->name-->WITH TEMP(noun)-->stat-->WITH-->
```

42. <u>expr</u>

```
          |--------------------------->|
          |                            |
--->sexpr--->VALUE--->sexpr--->expr op--->
```

43. <u>sexpr</u>

```
                      |----------------------------->|
         |->UPLUS--|  |                              |
-->term--|         |---->VALUE--->term--->sexpr op----->
         |->UMINUS-|
```

44. <u>term</u>

```
           |------------------------------->|
           |                                |
--->factor--->VALUE--->factor--->term op----->
                    |                    |
                    |<-------------------|
```

45. <u>factor</u>

```
--------------------------->name------------------------------>
   |                                                        |
   |--------------------->constant--------------------->|
   |                                                        |
   |------------------->function call------------------>|
   |                                                        |
   |------------------------>expr---------------------->|
   |                                                        |
   |------------------->factor-->NOT------------------->|
   |                                                        |
   |                                                        |
   |             |------------------------------------->|
   |             |                                          |
   |-->EMPTY SET----->expr-->INCLUDE--->END BUILDSET--->|
                   |                 |
                   |<----------------|
```

46. <u>function call</u>

```
-->name-->FUNCTION("type" noun)-->arg list-->CALL FUNC-->
```

47. <u>name</u>

```
-->VAR("var"noun,"type"noun)------------------------------>
 |                            |                   |   |
 |                            |<-----selection<------|   |
 |                            |                   |   |
 |                            |<----subscripting<----|   |
 |                            |                   |   |
 |                            |<-ARROW("type"noun)<--|   |
 |                                                    |
 |                                                    |
 |-------------->ROUTINE(noun)------------------------>|
```

47.1 selection

```
    |-->VCCMP("var" noun,"type" noun)---|
----|                                   |--->
    |-->RCCMP("routine" noun)-----------|
```

47.2 subscripting

```
-->ADDRESS-->expr-->SUB("index" noun,"element" noun)-->
```

49. constant

```
------>INDEX(value,"type" noun)--------->
  |                               |
  |--------->REAL(displ)--------->|
  |                               |
  |----->STRING(length,displ)---->|
```

OUTPUT FROM PASS 4

1. <u>program</u>

```
----> body ----> EOM(var length) ---->
```

29. <u>body</u>

```
BODY(mode,label,parm length,var length,stack length)--|
                                                      |
         |<-------------------------------------------|
         |
         |----->stat---->BODY END ---->
```

31. <u>stat</u>

```
----------------------------------->
   |                          |
   |<------ assignment <------|
   |                          |
   |<------ proc call <-------|
   |                          |
   |<------- if stat <--------|
   |                          |
   |<------ case stat <-------|
   |                          |
   |<----- while stat <-------|
   |                          |
   |<----- repeat stat <------|
   |                          |
   |<------ for stat <--------|
   |                          |
   |<------ with stat <-------|
```

32. <u>assignment</u>

```
             |->ADDRESS-------------------------->|
-->operand--|                                     |-->expr--|
             |->RESULT(displ,kind,noun,length)-->|          |
                                                             |
                                                             |
                         |<----------------------------------|
                         |
                         |--->STORE------->
```

33. <u>proc call</u>

```
----> operand ----> arg list ----> CALL PROC ---->
```

34. <u>arg list</u>

```
 |<-----------------------------------------------------------|
 |                                                            |
 |           |<---------------------------------------------||
 |           |                                              ||
--->ARG LIST-->expr---|                                     ||
                      |                                     ||
                      |                                     ||
|<--------------------|                                     ||
|                                                           ||
->CONST/VAR/SAVEPARM(mode,displ,context,kind,noun,length)-->
```

36. <u>if stat</u>

```
-->expr-->FALSE JUMP(11)-->stat------|
                                     |
    |<-------------------------------|
    |
    |  |->DEF LABEL(11)--------------------------->|
    |--|                                           |--->
       |->JUMP DEF(12,11)-->stat-->DEF LABEL(12)--|
```

37. <u>case stat</u>

```
                              |<-------------------------------|
                              |                                |
-->expr-->CASE JUMP(10)--->DEF LABEL(li)--|                    |
                                          |                    |
     |<-----------------------------------|                    |
     |                                 ¬                       |
     |                                 |                       |
     |-->CHKTYPE(kind,noun,length)--->stat-->JUMP(ln)---|
                                                    |
        |<------------------------------------------|
        |
        |-->CASE LIST(10,min,max,l1,...ln)----->
```

38. <u>while stat</u>

```
-->DEF LABEL(l1)-->expr-->FALSE JUMP(l2)-->stat---|
                                                  |
                    |<----------------------------|
                    |
                    |----->JUMP DEF(l1,l2)------>
```

39. <u>repeat stat</u>

```
--->DEF LABEL(l)--->stat--->expr--->FALSE JUMP(l)--->
```

40. <u>for stat</u>

```
-->operand-->ADDRESS-->expr---|
                              |
                              |
      |<----------------------|
      |
      |-->FOR STORE-->expr-->FOR LIM(l1,displ,op,l2)--|
                                                      |
                                                      |
           |<-----------------------------------------|
           |
           |-->stat-->FOR UP/DOWN(l1,l2)------->
```

41. <u>with stat</u>

```
-->operand-->ADDRESS-->WITH TEMP-->stat-->WITH-->
```

42. <u>expr</u>

```
          |--------------------------->|
          |                            |
--->sexpr--->VALUE--->sexpr--->expr op--->
```

43. <u>sexpr</u>

```
                             |------------------------------>|
         |->UPLUS--|         |                               |
-->term--|         |---->VALUE--->term--->sexpr op----->
         |->UMINUS-|                  |                |
                                      |<---------------|
```

44. term

```
          |-------------------------------->|
          |                                 |
--->factor--->VALUE--->factor--->term op----->
                   |                      |
                   |<-------------------- |
```

45. factor

```
------------------>operand------------------>
   |                                     |
   |----------->function call---------->|
   |                                     |
   |--------------->expr--------------->|
   |                                     |
   |----------->factor-->NOT----------->|
   |                                     |
   |--->EMPTY SET-->expr-->INCLUDE--->|
```

46. function call

```
-->operand-->FUNCTION(kind,noun,length)-->arg list---|
                                                     |
                                                     |
                                   |<----------------|
                                   |
                                   |
                                   |    |-->CALL FUNC--|
                                   |-->|               |--->
                                        |-->CALL GEN---|
```

47. <u>operand</u>

```
  |-->ROUTINE(mode,label,parm length,var length)-----|
--|                                                  |--->
  |-->VAR(mode,displ,context,kind,noun,length)-------|
                                                  |  |
                                                  |  |
                    |---------------------------->|  |
                    |                                |
                    |<---------selection<------------|
                    |                                |
                    |<--------subscripting<----------|
                    |                                |
                    |<---ARROW(kind,noun,length)<----|
```

47.1 <u>selection</u>

```
 |->VCOMP(mode,displ,context,kind,noun,length)-----------|
-|                                                       |->
 |->RCOMP(mode,label,parm length,var length,stack length)|
```

47.2 <u>subscripting</u>

```
--->ADDRESS-->expr-->SUB(min,max,length,"index" kind,
                         noun,length,"element" kind,
                         noun,length)------------------>
```

OUTPUT PASS 5

1. <u>program</u>

```
--> JUMP(1) --> body --> EOM(var length) -->
```

29. <u>body</u>

```
ENTER(mode,label,parm length,var length,temp length)---|
                                                       |
                                                       |
                                |<---------------------|
                                |
                                |-->stat-->RETURN(mode)------>
```

31. <u>stat</u>

```
---------------------------------->
   |                         |
   |<----- assignment <------|
   |                         |
   |<------ pro call <-------|
   |                         |
   |<------- if stat <-------|
   |                         |
   |<------ case stat <------|
   |                         |
   |<------ while stat <-----|
   |                         |
   |<----- repeat stat <-----|
   |                         |
   |<------ for stat <-------|
   |                         |
   |<----- with stat <-------|
```

32. assignment

```
                            |-->ASSIGN(type)--|
--->var addr-->expr---|                         |---->
                            |-->COPY(length)--|
```

33. proc call

```
----->arg list-->PROCEDURE(std number)-------------------->
  |                                                          |
  |-------->arg list-->CALL PROG-->POP(intf length)-->|
  |                                                          |
  |-->var addr-->FIELD(displ)--|                            |
  |                              |                            |
  |<---------------------------|                            |
  |                                                          |
  |-->arg list-->CALL(mode,label,parm length)-------->|
```

34. arg list

```
--->ARG LIST---------------->
              |           |
              |<--expr<--|
```

36. if stat

```
-->expr-->FALSE JUMP(l1)-->stat--->DEF LABEL(l1)-------->|
                                 |                          |
      |<-----------------------|                          |
      |                                                    |
      |->JUMP(l2)-->DEF LABEL(l1)-->stat-->DEF LABEL(l2)--->
```

37. <u>case stat</u>

```
                                   |<----------------------------|
                                   |                             |
->expr->CASE EXPR->JUMP(10)->DEF LABEL(li)->stat->JUMP(ln)--
                                                              |
                                                              |
    |<--------------------------------------------------------|
    |
    |->DEF LABEL(10)->CASE JUMP(min,max,l1,...ln)--|
                                                   |
                                                   |
                 |<--------------------------------|
                 |
                 |-->DEF LABEL(ln)----->
```

38. <u>while stat</u>

```
-->DEF LABEL(l1)-->expr-->FALSE JUMP(l2)-->stat--|
                                                 |
                                                 |
           |<------------------------------------|
           |
           |-->JUMP(l1)-->DEF LABEL(l2)----->
```

39. <u>repeat stat</u>

```
--->DEF LABEL(l)--->stat--->expr--->FALSE JUMP(l)--->
```

40. for stat

```
-->"control" var addr-->"initial" expr-->ASSIGN(type)--|
                                                       |
                                                       |
 |<----------------------------------------------------|
 |
 |->"limit" expr-->DEF LABEL(l1)-->"control" var value--|
                                                        |
                                                        |
 |<-----------------------------------------------------|
 |
 |->"limit" var value--|
                       |
                       |
        |<-------------|
        |
        |->COMPARE(ng/nl,word,l2 "exit",l1 "loop")--|
                                                    |
                                                    |
 |<-------------------------------------------------|
 |
 |->FALSE JUMP(l2)-->stat-->"control" var addr---|
                                                 |
                                                 |
 |<----------------------------------------------|
 |
 |->FOR STATEMENT(INCREMENT/DECREMENT(type)-->JUMP(l1)--|
                                                        |
                                                        |
                    |<----------------------------------|
                    |
                    |->DEF LABEL(l2)-->POP(word)----------->
```

41. with stat

```
---> var addr ---> stat ---> POP(word) --->
```

42. <u>expr</u>

```
         |---------------------------------------->|
         |                                         |
         |                                         |
         |             |-->COMPARE(comp,type)-------|   |
-->sexpr--->sexpr---|                            |----->
                    |-->COMPSTRUCT(comp,length)--|
```

43. <u>sexpr</u>

```
        |---------->|-------------------------------->|
        |           |                                 |
-->term--->NEG(type)---->term--->ADD/SUB/OR(type)-------->
                     |                            |
                     |<---------------------------|
```

44. <u>term</u>

```
           |--------------------------------------->|
           |                                        |
--->factor----->factor--->MUL/DIV/MOD/AND(type)------->
             |                                   |
             |<----------------------------------|
```

45. <u>factor</u>

```
---------------->PUSH CONST(value)---------------->
   |                                            |
   |--------------->variable--------------->|
   |                                            |
   |------------->function call------------->|
   |                                            |
   |--------------->expr----------------->|
   |                                            |
   |------------->factor--->NOT------------->|
   |                                            |
   |                                            |
   |              |----------------------->|
   |              |                             |
   |-->var value----->expr--->BUILD SET-----|
                    |                           |
                    |<----------------------|
```

46. <u>function call</u>

```
----->arg list--->FUNCTION(std number,type)----------->
  |                                            |
  |-->var addr--->FIELD(displ)--|              |
  |                             |              |
  |<----------------------------|              |
  |                                            |
  |-->FUNC VALUE(mode,type)--->arg list--|     |
                                         |     |
        |<-------------------------------|     |
        |                                      |
        |->CALL(mode,label,parm length)------>|
```

47. <u>variable</u>

```
    |--> var value --|
----|                |---->
    |--> var addr ---|
```

47.1 var value

```
     |-->PUSHVAR(type,mode,displ)---|
---->|                              |---->
     |-->var addr-->PUSHIND(type)---|
```

47.2 var addr

```
------->PUSHADDR(mode,displ)--------------------->
   |                                     |
   |                                     |
   |-->var addr------------------------->|
   |             |                  |    |
   |             |<---selection<----|    |
   |             |                  |    |
   |             |<--subscripting<--|    |
   |                                     |
   |                                     |
   |--->var value----------------------->|
```

47.3 selection

```
-----> FIELD(displ) ----->
```

47.4 subscripting

```
---> expr ---> INDEX(min,max,length) --->
```

OUTPUT PASS 6

1. program

```
--> JUMP(1) --> body --> EOM(var length) -->
```

29. body

```
ENTER(mode,label,parm length,var length,temp length)---|
                                                       |
                                                       |
                                  |<-------------------|
                                  |
                                  |-->stat-->RETURN(mode)------>
```

31. stat

```
---------------------------------->
   |                        |
   |<---- assignment <-----|
   |                        |
   |<----- pro call <------|
   |                        |
   |<------ if stat <------|
   |                        |
   |<----- case stat <-----|
   |                        |
   |<----- while stat <----|
   |                        |
   |<---- repeat stat <----|
   |                        |
   |<----- for stat <------|
   |                        |
   |<---- with stat <------|
```

32. assignment

```
                       |-->ASSIGN(type)--|
--->var addr-->expr---|                 |---->
                       |-->COPY(length)--|
```

33. proc call

```
----->arg list-->PROCEDURE(std number)-------------------->
  |                                                       |
  |-------->arg list-->CALLPROG-->POP(intf length)--->|
  |                                                       |
  |-->var addr-->FIELD(displ)--|                          |
  |                            |                          |
  |<---------------------------|                          |
  |                                                       |
  |-->arg list-->CALL(mode,label,parm length)-------->|
```

34. arg list

```
--->ARGLIST----------------->
               |           |
               |<--expr<--|
```

36. if stat

```
--->expR--->FALSEJUMP(l1)-->stat--->DEFLABEL(l1)-------->|
                                  |                        |
      |<--------------------------|                        |
      |                                                    |
      |--->JUMP(l2)-->DEFLABEL(l1)-->stat-->DEFLABEL(l2)--->
```

37. case stat

```
                                     |<------------------------------|
                                     |                               |
-->expr->CASEEXPR->JUMP(l0)-->DEFLABEL(li)->stat->JUMP(ln)--
                                                               |
                                                               |
     |<--------------------------------------------------------|
     |
     |-->DEFLABEL(l0)-->CASEJUMP(min,max,l1,...ln)--|
                                                    |
                                                    |
                  |<--------------------------------|
                  |
                  |-->DEFLABEL(ln)----->
```

38. while stat

```
--->DEFLABEL(l1)-->expr-->FALSEJUMP(l2)-->stat---|
                                                 |
                                                 |
            |<-----------------------------------|
            |
            |-->JUMP(l1)-->DEFLABEL(l2)----->
```

39. repeat stat

```
--->DEFLABEL(l)--->stat--->expr--->FALSEJUMP(l)--->
```

43. sexpr

```
          |------------------------------------------->|
          |                                            |
->term------->NEG(type)------->term-->ADD/SUB/OR(type)----->
       |   |                                         |  |
       |   |-->TRUEJUMP(lbl)-->term-->DEFLABEL(lbl)--->|  |
       |                                                |
       |<-----------------------------------------------|
```

44. <u>term</u>

```
          |--------------------------------------->|
          |                                        |
->factor------->factor---------->MUL/DIV/MOD/AND(type)----->
            | |                                  | |
            | |->FALSEJUMP(lbl)->factor->DEFLABEL(lbl)->| |
            |                                      |
            |<-------------------------------------|
```

45. <u>factor</u>

```
--------------->PUSHCONST(value)----------------->
   |                                    |
   |-------------->variable------------>|
   |                                    |
   |----------->function call---------->|
   |                                    |
   |--------------->expr--------------->|
   |                                    |
   |----------->factor--->NOT---------->|
   |                                    |
   |                                    |
   |             |--------------------->|
   |             |                      |
   |-->var value----->expr--->BUILDSET----->|
                   |                    |
                   |<-------------------|
```

46. <u>function call</u>

```
----->arg list--->FUNCTION(std number,type)----------->
  |                                         |
  |-->var addr--->FIELD(displ)--|           |
  |                              |           |
  |<-----------------------------|           |
  |                                         |
  |-->FUNCVALUE(mode,type)--->arg list---|   |
                                         |   |
        |<-------------------------------|   |
        |                                   |
        |->CALL(mode,label,parm length)------>|
```

47. variable

```
    |---> var value --->|
----|                   |---->
    |---> var addr ---->|
```

47.1 var value

```
     |-->PUSHVAR(type,mode,displ)--->|
---->|                               |---->
     |-->var addr-->PUSHIND(type)--->|
```

47.2 var addr

```
------->PUSHADDR(mode,displ)-------------------->
   |                                      |
   |--->var addr------------------------->|
   |              |                 |     |
   |              |<---selection<---|     |
   |              |                 |     |
   |              |<--subscripting<--|    |
   |                                      |
   |                                      |
   |--->var value------------------------>|
```

47.3 selection

```
----> FIELD(displ) ---->
```

47.4 subscripting

```
---> expr ---> INDEX(min,max,length) --->
```

OUTPUT PASS 7 and 8

1. <u>program</u>

```
--> JUMP(1) --> body --> EOM(var length) -->
```

29. <u>body</u>

```
ENTER(mode,label,parm length,var length,temp length)---|
                                                       |
                                                       |
                              |<-----------------------|
                              |
                              |-->stat-->RETURN(mode)------>
```

31. <u>stat</u>

```
---------------------------------->
   |                            |
   |<----- assignment <-----|
   |                            |
   |<------ pro call <------|
   |                            |
   |<------- if stat <------|
   |                            |
   |<------ case stat <-----|
   |                            |
   |<------ while stat <----|
   |                            |
   |<----- repeat stat <----|
   |                            |
   |<------ for stat <------|
   |                            |
   |<----- with stat <------|
```

32. assignment

```
                         |-->ASSIGN(type)--->|
--->var addr--->expr---|                     |------->
 |                       |-->COPY(length)--->|    |
 |                                                |
 |-------->ASSIGNIMM------>var addr----------->|
```

33. proc call

```
----->arg list-->PROCEDURE(std number)-------------------->
  |                                                      |
  |-------->arg list--->CALLPROG-->POP(intf length)-->|
  |                                                      |
  |-->var addr-->FIELD(displ)--|                         |
  |                            |                         |
  |<---------------------------|                         |
  |                                                      |
  |-->arg list-->CALL(mode,label,parm length)-------->|
```

34. arg list

```
--->ARGLIST----------------->
              |          |
              |<--expr<--|
```

36. if stat

```
--->expr-->FALSEJUMP(l1)--->stat--->DEFLABEL(l1)-------->|
                                  |                        |
      |<--------------------------|                        |
      |                                                    |
      |-->JUMP(l2)-->DEFLABEL(l1)--->stat-->DEFLABEL(l2)--->
```

37. <u>case</u> <u>stat</u>

```
                                       |<----------------------------|
                                       |                             |
-->expr->CASEEXPR->JUMP(l0)-->DEFLABEL(li)->stat->JUMP(ln)--
                                                              |
                                                              |
    |<--------------------------------------------------------|
    |
    |-->DEFLABEL(l0)-->CASEJUMP(min,max,l1,...ln)--|
                                                   |
                                                   |
                  |<-------------------------------|
                  |
                  |-->DEFLABEL(ln)----->
```

38. <u>while</u> <u>stat</u>

```
--->DEFLABEL(l1)-->expr-->FALSEJUMP(l2)-->stat---|
                                                 |
                                                 |
           |<------------------------------------|
           |
           |-->JUMP(l1)-->DEF LABEL(l2)----->
```

39. <u>repeat</u> <u>stat</u>

```
--->DEFLABEL(l)--->stat--->expr--->FALSEJUMP(l)--->
```

40. <u>for stat</u>

```
-->"control" var addr-->"initial" expr-->ASSIGN(type)--|
                                                       |
                                                       |
 |<----------------------------------------------------|
 |
 |-->"limit" expr-->DEFLABEL(l1)-->"control" var value--|
                                                        |
                                                        |
 |<-----------------------------------------------------|
 |
 |->"limit" var value--|
                       |
                       |
        |<-------------|
        |
        |->COMPARE(ng/nl,word,l2 "exit",l1 "loop")--|
                                                    |
                                                    |
 |<-------------------------------------------------|
 |
 |-->FALSEJUMP(l2)-->stat-->"control" var addr---|
                                                 |
                                                 |
 |<----------------------------------------------|
 |
 |-->FORSTATEMENT(INCREMENT/DECREMENT(type)-->JUMP(l1)--|
                                                        |
                                                        |
                   |<-----------------------------------|
                   |
                   |-->DEFLABEL(l2)-->POP(word)----------->
```

41. <u>with stat</u>

```
---> var addr ---> stat ---> POP(word) --->
```

42. expr

```
         |------------------------------------------>|
         |                                           |
         |                                           |
         |          |-->COMPARE(comp,type)-------|   |
-->sexpr--->sexpr---|                            |----->
                    |-->COMPSTRUCT(comp,length)--|
```

43. sexpr

```
      |---------------------------------------------->|
      |                                               |
->term------->NEG(type)---->term---->ADD/SUB(type)--------->
        | |               |                     | | |
        | |               |--->ADDIMM/          | | |
        | |                    SUBIMM/          | | |
        | |                    ORIMM(type)-->term--->| | |
        | |                                       | |
        | |--->TRUEJUMP(lbl)--->term--->DEFLABEL--->| |
        |                                           |
        |<------------------------------------------|
```

44. term

```
        |--------------------------------------------->|
        |                                              |
        | |-->MULIMM/DIVIMM/MODIMM/                    |
        | |                ANDIMM(type)--->factor--->| |
        | |                                          | |
->factor------>factor----->MUL/DIV/MOD(type)--------------->
         | |                                       | |
         | |->FALSEJUMP(lbl)->factor->DEFLABEL(lbl)->| |
         |                                           |
         |<------------------------------------------|
```

45. <u>factor</u>

```
----------------->PUSHCONST(value)---------------->
   |                                              |
   |--------------->variable---------------->|
   |                                              |
   |------------->function call------------->|
   |                                              |
   |----------------->expr------------------->|
   |                                              |
   |------------->factor--->NOT------------->|
   |                                              |
   |----------->NOTIMM--->factor------------>|
   |                                              |
   |                                              |
   |              |------------------------->|
   |              |                               |
   |-->var value----->expr--->BUILDSET------>|
                   |                              |
                   |<------------------------|
```

46. <u>function call</u>

```
              |--->FUNCTIONIMM(CHR/ABS/ORD/type)---->|
              |                                      |
--->arg list-->FUNCTION(std number,type)------------------->
 |                                                     |
 |--->var addr---->FIELD(displ)---|                    |
 |                                |                    |
 |<-------------------------------|                    |
 |                                                     |
 |                                                     |
 |-->
 |--->FUNCVALUE(mode,type)-->arg list---|              |
                                        |              |
                 |<---------------------|              |
                 |                                     |
                 |-->CALL(mode,label,parm length)------>|
```

47. <u>variable</u>

```
    |---> var value --->|
----|                   |---->
    |---> var addr ---->|
```

47.1 var value

```
     |-->PUSHVAR(type,mode,displ)-->|
---->|                              |---->
     |-->var addr-->PUSHIND(type)-->|
```

47.2 var addr

```
------->PUSHADDR(mode,displ)--------------------->
   |                                        |
   |---->var addr-------------------------->|
   |               |                    |   |
   |               |<----selection<-----|   |
   |               |                    |   |
   |               |<--subscripting<----|   |
   |                                        |
   |                                        |
   |--->var value-------------------------->|
```

47.3 selection

```
----> FIELD(displ) ---->
```

47.4 subscripting

```
---> expr ---> INDEX(min,max,length) --->
```

1. program

---> JUMP(loc,label) ---> body ---> EOM --->

29. body

---> enter ---> stat ---> return

29.1 enter

ENTERPROG(pop length,line,block,var length)
ENTERPROC(block,pop length,line,var length)

29.2 return

EXIT
EXITPROG

31. stat

```
----------------------------------->
   |                         |
   |<----- assignment <-----|
   |                         |
   |<------ proc call <-----|
   |                         |
   |<------ if stat <-------|
   |                         |
   |<------ case stat <-----|
   |                         |
   |<----- while stat <-----|
   |                         |
   |<----- repeat stat <----|
   |                         |
   |<------ for stat <------|
   |                         |
   |<------ with stat <-----|
```

32. assignment

```
----->var addr--->expr--->assign----------->
  |                                       |
  |--->assign immediate--->var addr--->|
```

32.1 assigin

POPFISINT
POPFISREAL
POPFISCHAR
POPFISSET
ASSIGNTAG
MOVESTRUC(length)

32. assign immediate

ASSIGNIMMINT
ASSIGNIMMCHAR

33. proc call

```
---> arg list ----->CALLSYS(number)--->|
                  |                    |--->
                  |-->CALL(loc,block)--->|
```

34. arg list

```
---------------->
   |         |
   |<--expr--|
```

36. if stat

```
                                         |------------------->|
                                         |                    |
-->expr-->FALSEJUMP(loc,l1)-->stat-->JUMP(loc,l2)-->stat-->
```

37. case stat

```
                         |<--------------------|
                         |                     |
-->expr-->JUMP(loc,lo)--->stat-->JUMP(loc,ln)----|
                                                 |
         |<--------------------------------------|
         |
         |---->BRANCHCASE(min,max-min,loc,l1...ln)---->
```

38. while stat

```
--->expr--->FALSEJUMP(loc,l2)--->stat--->JUMP(loc,l1)--->
```

39. repeat stat

```
---> stat ---> expr ---> FALSEJUMP(loc,1) --->
```

40. for stat

```
--->"control" var addr-->"initial" expr-->POPFISINT---|
                                                      |
                                                      |
  |<--------------------------------------------------|
  |
  |---->"limit" expr---->"control" var value----|
                                                |
                                                |
  |<--------------------------------------------|
  |
  |-->"limit" var value-->INTNGRFIS/INTNLSFIS---|
                                                |
                                                |
  |<--------------------------------------------|
  |
  |-->FALSEJUMP(loc,12)-->stat-->"control" var addr----|
                                                       |
                                                       |
  |<---------------------------------------------------|
  |
  |-->INCINT/DECINT-->JUMP(loc,11)-->POP(length)--------->
```

41. with stat

```
---> var addr ---> stat ---> POP(length) --->
```

42. expr

```
          |-------------------->|
          |                     |
--->sexpr---->sexpr---->expr op---->
```

42.1 expr op

INTLSFIS	REALLSFIS	CHARLSFIS	SETEQFIS
INTEQFIS	REALEQFIS	CHAREQFIS	SETINCALLFIS
INTGRFIS	REALGRFIS	CHARGRFIS	SETNEQFIS
INTNLSFIS	REALNLSFIS	CHARNLSFIS	SETWITHINFIS
INTNEQFIS	REALNEQFIS	CHARNEQFIS	SETMEMBFIS
INTNGRFIS	REALNGRFIS	CHARNGRFIS	
		CSSTRUC	NLSSTRUC
		EQSTRUC	NEQSTRUC
		GRSTRUC	NGRSTRUC

43. sexpr

```
       |---------------------------------------------->|
       |                                               |
-->term------>NEG(type)--->term-->sexpr op------------------->
         | |               |                        | | |
         | |               |-->sexpr imm op-->term-->| | |
         | |                                          | |
         | |--->TRUEJUMP(loc,lbl)--->term----------->| |
         |                                            |
         |<-------------------------------------------|
```

43.1 sexprop

ORFISINT
ORFISBOOL
ORFISSET
NEGINTFIS
NEGREALFIS
ADDFISINT
ADDFISREAL
SUBFISINT
SUBFISREAL

43.2 sexprimmop

ADDIMMINT
SUBIMMINT
ORIMMINT
EORIMMINT

44. term

```
              |---------------------------------------->|
              |                                         |
              |  |-->term imm op--->factor--->|         |
              |  |                            |         |
--->factor------->factor---->term op-------------------------->
               | |                               | |
               | |--->FALSEJUMP(loc,lbl)-->factor-->| |
               |                                   |
               |<----------------------------------|
```

44.1 termop

ANDFISINT
ANDFISBOOL
ANDFISSET
MULINTFIS
MULREALFIS
DIVINTFIS
DIVREALFIS
MODINTFIS

44.2 term imm op

MULIMMINT
DIVIMMINT
MODIMMINT
ANDIMMINT

45. factor

```
------->PUSHINTFIS------------------------------>
   |                                        |
   |--->PUSHREALFIS------------------------>|
   |                                        |
   |--->PUSHSETFIS------------------------->|
   |                                        |
   |--->PUSHCHARFIS------------------------>|
   |                                        |
   |--->variable--------------------------->|
   |                                        |
   |--->function call---------------------->|
   |                                        |
   |--->expr------------------------------->|
   |                                        |
   |--->factor--->NOT---------------------->|
   |                                        |
   |                                        |
   |              |------------------------>|
   |              |                         |
   |--->var value----->expr--->BUILDSET---->|
                    |                    |
                    |<-------------------|
```

46. function call

```
------>arg list----->std func------------------------------>
 |                                                        |
 |                                                        |
 |                                |-->CALLSYS(number)--|  |
 |->FUNCVALUE(mode)-->arg list--|                      |-->|
                                |-->CALL(loc,block)--|
```

46.1 STD FUNC

ABSINTFIS
ABSREALFIS
CNVTINTFIS
CNVTCHARFIS
INCINTFIS
SUCCCHARFIS
DECINTFIS
PREDCHARFIS
CNVTREALFIS

47. variable

```
     |---> var value --->|
---->|                   |---->
     |---> var addr ---->|
```

47.1 var value

```
--------> PUSHINTFIS --------------->
   |                               |
   |----> PUSHREALFIS ----------->|
   |                               |
   |----> PUSHCHARFIS ----------->|
   |                               |
   |----> PUSHSETFIS ------------>|
```

47.2 var addr

```
-----------> PUSHADDR -------------------------->
   |                                         |
   |                                         |
   |--> var addr ------------------------->|
   |               |                   |     |
   |               |<--selection<-----|     |
   |               |                   |     |
   |               |<--subscripting<--|     |
   |                                         |
   |                                         |
   |-----> var value ------------------->|
```

47.3 selection

```
    ADDINTFIS
```

47.4 subscripting

```
--->expr--->INDEX(min,dimension,length)--->
```

1. program

```
--->(prog length,code length,stalk length,var length)--->|
                                                          |
     |<---------------------------------------------------|
     |
     |--->JUMP(disp)--->body--->constants------>
```

29. body

```
---> enter ---> stat ---> return
```

29.1 enter

```
ENTERPROG(pop length,line,stalk length,var length)
ENTERPROC(block,pop length,line,var length)
```

29.2 return

```
EXIT
EXITPROG
```

31. <u>stat</u>

```
---------------------------------->
   |                         |
   |<----- assignment <-----|
   |                         |
   |<------ proc call <-----|
   |                         |
   |<------ if stat <-------|
   |                         |
   |<------ case stat <-----|
   |                         |
   |<----- while stat <-----|
   |                         |
   |<----- repeat stat <----|
   |                         |
   |<------ for stat <------|
   |                         |
   |<------ with stat <-----|
```

32. <u>assignment</u>

```
----->var addr--->expr--->assign----------->
  |                                     |
  |--->assign immediate--->var addr--->|
```

32.1 <u>assigin</u>

POPFISINT
POPFISREAL
POPFISCHAR
POPFISSET
ASSIGNTAG
MOVESTRUC(length)

32.2 <u>assign immediate</u>

ASSIGNIMMINT
ASSIGNIMMCHAR

33. <u>proc call</u>

```
---> arg list ----->CALLSYS(number)--->|
                |                      |--->
                |-->CALL(disp)-------->|
```

34. <u>arg list</u>

```
---------------->
   |         |
   |<--expr--|
```

36. <u>if stat</u>

```
                                   |--------------------->|
                                   |                      |
-->expr-->FALSEJUMP(disp1)-->stat--->JUMP(disp2)-->stat--->
```

37. <u>case stat</u>

```
                       |<-----------------------|
                       |                        |
-->expr-->JUMP(disp0)--->stat-->JUMP(dispn)------|
        |<---------------------------------------|
        |
        |--->BRANCHCASE(min,max-min,disp1...dispn)--->
```

38. <u>while stat</u>

```
--->expr--->FALSEJUMP(disp2)--->stat--->JUMP(disp1)--->
```

39. repeat stat

```
---> stat ---> expr ---> FALSEJUMP(disp) --->
```

40. for stat

```
--->"control" var addr-->"initial" expr-->POPFISINT---|
                                                      |
                                                      |
  |<--------------------------------------------------|
  |
  |---->"limit" expr---->"control" var value----|
                                                |
                                                |
  |<--------------------------------------------|
  |
  |-->"limit" var value-->INTNGRFIS/INTNLSFIS---|
                                                |
                                                |
  |<--------------------------------------------|
  |
  |-->FALSEJUMP(disp2)-->stat-->"control" var addr----|
                                                      |
                                                      |
  |<--------------------------------------------------|
  |
  |-->INCINT/DECINT-->JUMP(disp1)-->POP(length)--------->
```

41. with stat

```
---> var addr ---> stat ---> POP(length) --->
```

42. expr

```
          |-------------------->|
          |                     |
--->sexpr---->sexpr---->expr op---->
```

43. sexpr

```
        |---------------------------------------------->|
        |                                               |
-->term------>NEG(type)--->term-->sexpr op----------------->
         | |              |                        | | |
         | |              |-->sexpr imm op-->term-->| | |
         | |                                        | |
         | |------>TRUEJUMP(disp)--->term---------->| |
         |                                            |
         |<-------------------------------------------|
```

44. term

```
           |------------------------------------------>|
           |                                           |
           |  |-->term imm op--->factor--->|           |
           |  |                            |           |
--->factor------->factor---->term op--------------------->
             | |                            | |
             | |---->FALSEJUMP(disp)--->false---->| |
             |                                    |
             |<-----------------------------------|
```

44.1 termop

ANDFISINT
ANDFISBOOL
ANDFISSET
MULINTFIS
MULREALFIS
DIVINTFIS
DIVREALFIS
MODINTFIS

44.2 term imm op

MULIMMINT
DIVIMMINT
MODIMMINT
ANDIMMINT

45. factor

```
------->PUSHINTFIS--------------------------------->
    |                                            |
    |--->PUSHREALFIS--------------------------->|
    |                                            |
    |--->PUSHSETFIS---------------------------->|
    |                                            |
    |--->PUSHCHARFIS--------------------------->|
    |                                            |
    |--->variable------------------------------>|
    |                                            |
    |--->function call------------------------->|
    |                                            |
    |--->expr---------------------------------->|
    |                                            |
    |--->factor--->NOT------------------------->|
    |                                            |
    |                                            |
    |               |-------------------------->|
    |               |                            |
    |--->var value----->expr--->BUILDSET----->|
                     |                    |
                     |<-------------------|
```

46. function call

```
----->arg list----->std func--------------------------------->
 |                                                           |
 |                                                           |
 |                              |-->CALLSYS(number)--|       |
 |->FUNCVALUE(mode)-->arg list--|                    |-->|
                                |-->CALL(disp)-------|
```

46.1 STD FUNC

ABSINTFIS
ABSREALFIS
CNVTINTFIS
CNVTCHARFIS
INCINTFIS
SUCCCHARFIS
DECINTFIS
PREDCHARFIS
CNVTREALFIS

47. variable

```
     |---> var value --->|
---->|                   |---->
     |---> var addr ---->|
```

47.1 var value

```
--------> PUSHINTFIS --------------->
   |                               |
   |----> PUSHREALFIS ----------->|
   |                               |
   |----> PUSHCHARFIS ----------->|
   |                               |
   |----> PUSHSETFIS ------------>|
```

47.2 var addr

```
-----------> PUSHADDR ------------------------->
   |                                      |
   |                                      |
   |--> var addr ----------------------->|
   |              |                 |    |
   |              |<--selection<-----|    |
   |              |                 |    |
   |              |<--subscripting<--|    |
   |                                      |
   |                                      |
   |-----> var value ------------------->|
```

47.3 selection

ADDINTFIS

47.4 subscripting

```
--->expr--->INDEX(min,dimension,length)--->
```

APPENDIX C:

USER'S MANUAL FOR PASCAL/S

APPENDIX C

There are numerous options available to the programmer in the PASCAL/S version of the compiler. This appendix provides a list of these options. The list summarizes the available options, by including a brief description of the option, where it is used, how it is implemented, and the default for the option.

The list of compiler options must precede the first executable line of code in the program. The list must be enclosed in parentheses with the entries separated by commas. Pass 1 scans the list immediately after pass initialization and sets the options in the argument list. Only the first character of an option identifier is scanned except in the case of integer type options. In this case, the integer option identifier is scanned to determine the length (2, 4, or 8).

Compiler options are communicated to all passes through an argument list. The argument list is part of the interpass record which remains in the heap between passes during compilation.

The options which are currently implemented in the PASCAL/S version of the compiler are listed in the table.

OPTION	DEFAULT	USEAGE
		COMMENTS
LISTOPTION	off	pass 10
		produces object listing. Turned on by 'TEST' or 'LIST' in options list.
SOURCEOPTION	on	pass 1
		produces object listing. Turned on by 'TEST' and off by 'SOURCENO' in options list.
TESTOPTION	off	all passes
		Prints intermediate code. Turned on by 'TEST' in options list.
CHECKOPTION	on	pass 9
		Generates range and pointer checks in object code. Turned off by 'CHECKNO' in options list.
CODEOPTION	off	passes 9 & 10
		Set by pass 9 if there are no errors. Pass 10 generates object code if CODEOPTION is set. This option can not be set by the user.

NUMBEROPTICN on pass 9

Generates line numbers in the object code. Turned off by 'NUMBERNO' in the options list.

OPTIMOPTICN off Passes 7,8,9

Includes optimization. Turned on by 'OPTIMIZATION' in options list.

ASSEMOPTICN on pass 10

Generates object code. Turned off by 'ASSEMNO' in options list.

INT2OPTION on pass 1 *

Indicates length in bytes of integers. Turned on by 'INT2' in options list.

INT4OPTION off pass 1 *

Indicates length in bytes of integers. Turned on by 'INT4' in options list.

INT8OPTION off pass 1 *

Indicates length in bytes of integers. Turned on by 'INT8' in options list.

XREFOPTION off XREF pass

Initiates Cross Reference. Turned on by 'XREF' in options list.

DUMP off passes 5a,6a,7a,8a

Writes intermediate code in a readable form, i.e. formatted mnemonic representation. Turned on by 'DUMP' in the options list.

* Only one INT option may be in effect at any one time. The last one listed in the options list or the default, INT2, is the one used.

While pass 1 test and sets the options, the options are tested in the passes indicated in the useage column of the table except for three options. They are OPTIMIOPTION, XREFOPTION and DUMP. Since the occurrence of one of these three necessitates the initiation of an additional pass or passes, the options list is checked by the driver. Thus, the driver retrieves the options from the argument list and determines the correct calling sequence.

Passes 5a, 6a, 7a, and 8a referred to in the DUMP option represent calls to additional passes which follow passes 5, 6, 7, and 8 respectively. Each of the 'a' passes takes as input the intermediate code from the previous pass and writes it in readable form leaving the intermediate code unchanged. The output is a formatted version of the intemediate code with the mnemonic description of all the intermediate code integers.

PASCAL/S:
SEQUENTIAL PASCAL
WITH DATA TYPE EXTENSIONS

BY

BARBARA K. NORTH

B.S., KANSAS STATE UNIVERSITY, 1970
M.S., KANSAS STATE UNIVERSITY, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1978

This report documents a ten-pass optimizing compiler for the programming language Sequential Pascal (with extensions). This version of the compiler, denoted PASCAL/S, was developed for implementation on a specific stack machine architecture. Further, this compiler was adapted from the compiler for the programming language Concurrent Pascal which was written and documented by Alfred C. Hartmann. Following brief introductions to Hartmann's compiler and the PASCAL/S compiler, the major contributions of the new version are described in detail. The language modifications and extensions which have been added are enumerated. Three of the ten passes of the new version are designated as optimizing passes. The optimization performed in this compiler includes constant folding, Boolean and arithmetic expression evaluation and miscellaneous other optimization. This optimization is described and numerous examples are provided. The process of code generation is also described. The main distinction between code generation of the current version and Hartmann's version is in the code that is being selected. In the PASCAL/S version the code selected in the next-to-last pass is being selected for the specific stack machine. This document includes the following appendices: the SEQUENTIAL PASCAL REPORT as modified to reflect the changes in the PASCAL/S version; the syntax graphs depicting the eleven languages associated with this version; and a User's Manual which contains the various user options.