A Pseudo-Machine
For Code Generation

by

Henry John Pasko

Technical Report CSRG - 30
December 1973

COMPUTER SYSTEMS RESEARCH GROUP
UNIVERSITY OF TORONTO
A Pseudo-Machine
For Code Generation

by

Henry John Pasko

Technical Report CSRG - 30
December 1973
ABSTRACT

The purpose of this thesis is to investigate the practicality of the use of pseudo-machines for code generation. This approach essentially divides the translation of a source language program (PASCAL in this case) to a real machine into two steps. The first step is independent of the target machine; it compiles the program into a machine language of an imaginary machine. The second step then takes this imaginary machine code and compiles it into machine code for a real machine. (Work has been completed which implements the machine independent first step, and it is now possible to utilize this compiler in constructing a set of PASCAL translators for different machines.)

The design and implementation of a language translator for the PASCAL language using this approach are presented. A discussion of pseudo-machines and related work is presented and the language PASCAL is then briefly described. A detailed explanation of the architecture and instruction set of the PASCAL pseudo-machine follows. The implementation of a compiler is then described. In conclusion, the results of this investigation, as well as the effect on machine independence, portability and overall efficiency are discussed.
A PSEUDO-MACHINE
FOR CODE GENERATION

by

Henry John Pasko

A thesis
submitted in partial fulfillment
of the requirements
for the degree of
Master of Science

Department of Computer Science
University of Toronto
To My Parents
ACKNOWLEDGEMENTS

I would like to express my appreciation and thanks to Professor R. C. Holt for his guidance, supervision, many helpful suggestions and encouragement during the work on this thesis.

Professors R. Schild and D. B. Wortman also deserve special thanks for reading the thesis and suggesting improvements.

I would also like to thank Mr. B. L. Clark for his valuable suggestion of the range table which was much appreciated.

U. Ammann and K. Jensen, whose compiler-interpreter system for PASCAL provided many ideas for this thesis, also deserve special thanks. I also wish to express my appreciation to Mr. L. Kerr and Mr. R. Camm for providing access to this compiler-interpreter system.

Finally, I would like to thank the National Research Council of Canada and the Department of Computer Science whose funding made this project possible.
# TABLE OF CONTENTS

## INTRODUCTION

1.1 Pseudo-Machines........................................ 3

1.2 Related Work........................................... 5
  1.2.1 Interpreters......................................... 5
  1.2.2 Source Language Machines and Micro-Programmed Machines........... 7

1.3 Pseudo-Machines for Code Generation.............. 8
  1.3.1 Approach........................................... 8
  1.3.2 Merits of Approach.................................. 8

1.4 The PASCAL Language................................. 10
  1.4.1 Aims of the Language................................ 10
  1.4.2 Language Description............................... 10

## THE PASCAL PSEUDO-MACHINE

2.1 Overview of the PASCAL Pseudo-Machine........... 13

2.2 The Architecture of the PASCAL Pseudo-Machine.... 15

2.3 The PASCAL Pseudo-Machine Language.............. 20
  2.3.1 Locating and Fetching Variables and Constants.............. 20
  2.3.2 Storing and Transfer of Values to Variables.............. 22
  2.3.3 Function and Procedure Linkage....................... 24
  2.3.4 Expressions....................................... 27
  2.3.5 Statements........................................ 34
  2.3.6 Standard Functions and Procedures.................... 43
3  COMPILER STRUCTURE

3.1 General Overview

3.2 The Structure of the Compiler
   3.2.1 The Symbol Table
   3.2.2 The Type Stack
   3.2.3 The Constant Stack
   3.2.4 Exit Label List and Label List... 
   3.2.5 The Declaration and Pseudo-Machine Tables
   3.2.6 The Organization of the Compiler

3.3 Summary

4  CONCLUSIONS

4.1 Feasibility of Approach to Code Generation...

4.2 Effect of Using a Pseudo-Machine
   for Code Generation
   4.2.1 On Machine Independence
   4.2.2 On Compiler Portability
   4.2.3 On Efficiency

REFERENCES

APPENDICES

A  Syntax Charts for PASCAL

B  PASCAL Pseudo-Machine Instructions

C  Code Generation for the IBM S/370

D  Unsupported Features of PASCAL

E  Subset of SUE Utilized
CHAPTER 1

INTRODUCTION

1.1 Pseudo-Machines

The trend today, in the area of computer programming, is towards the development and utilization of higher level languages. Languages such as FORTRAN, COBOL, PL/I and ALGOL are enjoying widespread popularity in preference to the once popular machine level languages. The reasons for this popularity can be attributed to their comparative ease of use and to the fact that programs coded in higher level languages are less prone to error than machine language programs. Higher level languages allow the programmer to focus his attention on the problem he is solving, rather than on the details of the machine language.

However, this development in the area of programming languages has not been matched by a similar development in the architecture of computer hardware, for which language translators for these languages must be written. The language translator writer is faced not only with von Neumann type hardware, (that is, without stacks), which is not ideally suited for many of these languages, but also with a lack of hardware standardization from manufacturer to manufacturer, and even from model to model from the same manufacturer. One current solution to this problem is that a different language translator be written for each type of machine the language is to run on. However, it is immediately obvious that this approach is not very satisfactory. It is very time-consuming, inefficient and quite often, certain features of the language under consideration are either implemented inefficiently, or are not implemented at all. Another solution is to design a language and write its compiler for only one machine, and ignore the rest. This approach is even less pleasing than the previous one.

An approach becoming popular today, which solves part of these problems for the translator writer, is the use of a pseudo-machine. A pseudo-machine for a high level language is a hypothetical computer, the architecture of which strongly reflects the nature of the language for which it is being used to implement. The instructions of this machine are usually fairly sophisticated and represent many of the constructs within the higher level language. This pseudo-
machine now greatly simplifies the task of the translator writer.

Once this pseudo-machine has been designed, a compiler is written for the higher level language. This compiler emits code from source programs of the higher level language in the instruction set of the pseudo-machine. If the compiler is now rewritten in the language which it compiles, the compiler now becomes portable and machine independent. Unfortunately no real machine is capable of directly executing the pseudo-machine code.

Several approaches, discussed in the next sections, are now possible. An interpreter may be written on an existing machine in an existing language. An actual machine may be built which executes the pseudo-machine code. An existing machine may be micro-programmed to execute the pseudo-machine code. Or, a compiler may be written which compiles this pseudo-machine code and emits machine code for a real machine. This final approach is the subject of this thesis.
1.2 Related Work

Before proceeding with an examination of pseudo-machines for code generation, let us examine some previous work in the area of pseudo-machines.

The idea of the use of pseudo-machines for translator writing is not recent. One of the original design papers on pseudo-machines was presented by Grau [Grau 1962]. In this paper, he presents a discussion of a pseudo-language, called L. The language was designed to execute on a stack machine. It was also demonstrated that this language could be used for translation for ALGOL and ALGOL-like languages.

1.2.1 Interpreters

One approach to executing the code produced for the pseudo-machine, is to write an interpreter. A program is written in an existent language, which simulates the actions of the pseudo-machine on a real machine. One of the earliest efforts in this direction was an intermediate language machine for ALGOL 60 written by van der Poel and simulated on the Zebra computer [van der Poel 1962].

Another of the early efforts was Randell and Russell's ALGOL 60 machine [Randell and Russell 1964] written for the KDF-9 computer. This work was the first extensive treatment of the problems of implementing ALGOL 60 on the computers available at that time.

In 1968, Glass [Glass 1968] reported on SPLINTER, an interpreter written in FORTRAN for PL/I to run on the UNIVAC 1108. The SPLINTER system emphasized debugging and diagnostic capabilities, exploiting the interpretive design to supply features difficult to obtain through traditional compilation-execution techniques.

Further efforts were made towards the construction of intermediate machines for PL/I. A pseudo-machine first described by Goodfellow [Goodfellow 1968] was later extended by Pullam [Pullam 1969]. This machine features many address instructions, the use of self describing (typed) data and the use of stacks for storage.

Sugimoto [Sugimoto 1969] also describes a PL/I machine. This machine is based on a list structured machine organization and uses typed data and multiple address instructions.
Wortman [Wortman 1972] presents an interpreter for a pseudo-machine for a dialect of PL/I he calls Student PL. The main feature of this effort is that the use of scientific experimentation as a tool is examined for the design and evaluation of language directed computers. An initial design for a pseudo-machine is presented, followed by the experimental techniques used to measure the efficiency of this model. Based upon these measurements, a modified and improved pseudo-machine is obtained which is tailored more specifically to the language for which it is used.

Finally, another pseudo-machine designed for PL/I appears in [Boulton and Jeanes 1972]. The main feature of this system, called the PLUTO system, is its extensive diagnostic and dynamic debugging facilities, made possible by the use of a pseudo-machine.

A recent language translator system has also been developed for processing the language PASCAL [Jensen 1973a], [Jensen 1973b] and [Ammann 1973]. The compiler was written in PASCAL and was translated into the intermediate machine code used by this system. This has resulted in portability for PASCAL, in that only an interpreter need be written for the intermediate machine language, in order that one have a working PASCAL compiler.
1.2.2 Source Language Machines and Micro-Programmed Machines

The utilization of these machines for the execution of pseudo-machine code has not been as extensively investigated as interpreters. Perhaps the reason for this is the fact that machines are more difficult to design and micro-program than they are to program. However, several attempts have been made, and these are reported below.

In 1961, Anderson [Anderson 1961] presented the architecture of a stack machine to execute ALGOL 60. Another stack machine, designed for the execution of FORTRAN statements, was presented by Melbourne and Pugmire [Melbourne and Pugmire 1965]. This machine was micro-programmed on another machine. Another FORTRAN machine was presented in 1967 [Bashkow et al 1967]. This machine was designed to read in a FORTRAN source program, load it into core, almost as is, and execute it.

Weber [Weber 1967], describes a micro-programmed system which implements EULER on the IBM S/360 Model 30. This system consists of a translator which is a one-pass, syntax-driven compiler which translates EULER source language programs into a reverse Polish string form, and an interpreter, written in micro-code, which interprets the string language programs. It was demonstrated that an interpretive language can be executed efficiently by micro-programs on existing hardware.

The design of hardware for a machine capable of executing APL was presented in 1970 [Thurber and Myrna 1970]. A parallel processor was featured for matrix and vector operations. This machine was, for the major part, implemented via hardware, the remainder being micro-programmed. Abrams [Abrams 1970] presented a more complete design of an APL machine which was based on a detailed mathematical analysis of the properties of APL. One of his principal results was a method for replacing vector and array operations by transformations on their descriptors. Another hardware implemented language, SYMBOL, is described by Chesley and Smith [Chesley and Smith 1971].

More recently, the IBM S/360 Model 25 has been used for a micro-programmed implementation of APL [Hassitt et al 1973]. APL statements and functions are translated into an internal format which closely resembles the original. These statements and functions are then executed by a micro-programmed interpreter.

Finally, the Burroughs series of machines, beginning with the B5500, have been among the most successful "Algol" machines. The principal feature of these machines is their hardware stack which facilitates the task of implementing Algol-like languages.
1.3 Pseudo-Machines for Code Generation

1.3.1 Approach

The aim of this thesis is to investigate the use of pseudo-machines for code generation. This idea is not new, in that any multi-pass language translator emitting code for a particular machine represents the original source program in some intermediate internal form or code for its successive passes. For example, we have the Source Language Machine of Wilcox [Wilcox 1971]. However, the emphasis in this thesis is on machine independence of the intermediate pseudo-machine and portability of the compiler which produces it.

A language translator written using this technique now becomes divided into two distinct phases, as shown in Figure 1-1. The first, which we shall call the compiler step, performs all of the syntax and semantic checking for a source program. All variables are replaced by address couples, and statements are translated into the code of a pseudo-machine. Very briefly stated, the compiler removes all symbolic information (such as variable names) from the source program it compiles. In certain instances, where many data types are involved in the source program, tables are also generated which hold the type information for all the variables in the program. However, no storage is allocated for variables. Storage allocation is a machine dependent operation, and is left for the second phase.

The second phase is called the code generation step. The code and tables generated by the compiler are input into this phase and code is emitted for a real machine. This step of the translator is written for each type of machine for which code is to be emitted.

1.3.2 Merits of Approach

There are several advantages to this approach to translator writing. One is that the compiler for the language for which the pseudo-machine has been designed becomes portable and machine-independent. The pseudo-machine does not rely upon the quirks of specific hardware.
Figure 1-1 Phases of a Language Translator Using A Pseudo-Machine For Code Generation

- High Level Language Program
- Range Table
- Pseudo-Machine Code
- Real Machine Code
Implementation of a translator for this language on more than one machine becomes much easier because the use of a pseudo-machine allows a bootstrapping process. A compiler is written for the higher level language in an already existent language. The compiler is then rewritten in the language which it compiles and the first compiler compiles the rewritten version into the code of the pseudo-machine. The only step which requires a complete rewrite, is the code generation step. The first step remains invariant from machine to machine.

This type of implementation strategy is more efficient from the standpoint of problem program execution and storage allocation than the interpreter approach. The problem program is transformed into real machine code, which can execute more quickly than an interpreter. And the natural storage boundaries of the machine can be used, allowing for the compacting of storage.

However, a slight disadvantage in this approach, when compared to a one-step translator, appears in the fact that two steps must be executed to compile any program of the higher level language. Hence a higher overhead may be incurred than with a translator which goes directly to the code of a real machine. Also, an interpreter usually is simpler to program than a code generator.

Nonetheless, it is felt that the advantages of this approach to translator writing far outweigh any disadvantages.
1.4 The PASCAL Language

We have applied this approach of employing a pseudo-machine for code generation to producing a translator for the language PASCAL. The results obtained in this thesis are also applicable in constructing a language translator for the SUE System Language [Clark 1971]. This is the case because of the similarity of the two languages. Before presenting the details of this work, (the subject of following chapters), a brief discussion is now presented on the language PASCAL.

1.4.1 Aims of the Language

Due to his deep dissatisfaction with present day major languages, Wirth argues, in his Revised Report on PASCAL, that their "... features and constructs too often cannot be explained logically and convincingly and ... too often represent an insult to minds trained in systematic reasoning." [Wirth 1973]. Feeling that the disorder governing these languages imposes itself upon one's thinking, Wirth set forth two aims upon which he based the development of PASCAL. As stated by Wirth, these are:

(1) " ... to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language"

(2) " ... to develop implementations of this language which are both reliable and efficient on presently available computers".

1.4.2 Language Description

The following paragraphs comprise a very brief description of the PASCAL language with its data and control structures. For a more complete description, see [Wirth 1973]. In presenting examples of PASCAL programs, we have adopted the capitalization rules of the SUE System Language [Clark 1971]. Very briefly, these state that all names
invented by the programmer appear with the first letter capitalized.

1.4.2.1 Program Structure

A program in the PASCAL language consists of a program block. This program block consists of a sequence of declarations which define labels, manifest constants, types and variables. This list is followed by a sequence of procedure and function definitions. Finally, these definitions are followed by a list of statements which make up the body of the program. Procedures and functions are defined recursively as program blocks.

1.4.2.2 Data Structures

Several basic data types exist in the language. These are integer, real, character, and boolean. The programmer also has the ability to define his own data types by specifying the symbolic names of all constants belonging to that type. For example:

\[
\text{type Operator} = \text{(Plus, Minus, Multiplies, Divides)};
\]

Structured types are available in the language. These include arrays, records, pointers, sets and files. These structured data types are constructed using the above four basic types plus programmer types as the element types. More complex data structures can be constructed recursively.

File data types are structures consisting of a sequence of components which are all of the same type. Associated with the file is a buffer variable which contains the last component input or the component to be output. Files cannot be defined recursively.

Variables may be declared to be of type \(<\text{constant}\> .. \(<\text{constant}\). This defines a subrange of values which the variable may validly assume.
The control constructs of PASCAL allow both repetition and selection of statements. Repetition can be controlled by tests at either the beginning or end of the sequence of statements. The while <expression> do <statement> construct allows testing at the beginning of the sequence, whereas repeat <statement list> until <expression> allows testing at the end of the sequence. Bounded repetition is provided by the construct for <control variable> := <expression1> to <expression2> do <statement>.

Selection is provided by the case and if then else constructs. The case construct selects a particular statement from a list of statements for execution. Each statement is labelled by at least one constant of the type of the expression in the case header. The if then else construct provides selection capability for a boolean expression.

Finally, unconditional jumps are made possible by the inclusion of the goto construct in the language.
2.1 Overview of the PASCAL Pseudo-Machine

In this chapter, the architecture of an intermediate pseudo-machine used to implement PASCAL is described. The principal criterion used in the design of this pseudo-machine, was to facilitate code generation for real, commercial machines which exist to-day (for example, the IBM S/370 and PDP-11). The initial design was based on a compiler-interpreter written by U. Ammann and K. Jensen, [Ammann 1973], [Jensen 1973a] and [Jensen 1973b]. Their compiler and pseudo-machine have been taken and modified to suit the purposes of code generation. The major modifications included alteration of the method of storage allocation for variables and the method for accessing them, and the addition of a range table to the output of the compiler. In order to produce a PASCAL compiler which emits code for our pseudo-machine, we decided to use as much of the existing PASCAL compiler as possible. This was done as a time-saving measure.

A description of the architecture of the intermediate PASCAL machine is presented in detail in the following sections. Each component and its role in the pseudo-machine are described. Finally, the ideal machine language is introduced and the code sequence emitted for each PASCAL statement is given.

The semantics of each instruction of the pseudo-machine are discussed independent of PASCAL, although we do not intend that these instructions be executed. We felt that this was the most convenient means of describing the pseudo-machine language. Also, by the presenting these semantics, one may define correctness of a program written in the pseudo-machine language independent of its representation in PASCAL. Hence, a proof of correctness for the compiler in its transformation of a PASCAL source language program to a pseudo-machine language program is now possible. If a PASCAL source program and the corresponding pseudo-machine language program produced by a compiler yield the same results, then the compiler was correct in its translation of that program. This problem is not investigated in this thesis.

Throughout the following discussion, intermediate machine and pseudo-machine are used interchangeably. The compiler and compilation step refer to the first step of the
language translator and the code generator and the code generation step refer to the second step.
This section describes each component of the ideal pseudo-machine illustrated in figure 2-1. The overall structure of the machine is that of a so-called "Algol-Machine"; that is, the predominant feature of the machine is its reliance upon the stack. This basic design was chosen because of the block structure and recursive nature of PASCAL. There are four major stacks, three of which have their own displays which delimit or mark off the various lexic levels within the stacks. There are also two tables, one of which contains the pseudo-machine code to be executed or translated to "real" machine code. The other contains range information of the various variables used throughout the executing program. Finally, two registers contain status information about the executing program.

The Pseudo-Machine Code Table contains the PASCAL pseudo-machine code emitted for the PASCAL program. This code consists of PASCAL pseudo-machine instructions with the following format:

```
label op-code operand1,operand2
```

In the above instruction, the label field may or may not appear, depending upon whether a branch is made to this instruction. The op-code is the mnemonic of an instruction of the PASCAL ideal machine. One or both of the operands of this instruction, operand1 and operand2, may be missing depending upon the instruction.

The Run Stack contains the block marks for the currently active procedures and functions. Each time a new scope block is entered, a block mark with the following information is pushed onto this stack:

```
return address
save for displays
statement register save
dynamic link
static link
result (function)
```

This record contains the status information of the calling program. This includes the address of the first instruction to be executed after control returns from the called procedure (return address), the status of the various displays at invocation time (save for displays), the contents of the statement register (statement register save), and the dynamic and static links (dynamic link and static link). The dynamic link points back to the invoking procedure; the static link points to the scope level immediately surrounding the invoked procedure. The result field is used for function invocations; the result of the function is stored in this field. Note that the returned
Figure 2-1 The PASCAL Pseudo-Machine
result of a function in PASCAL may only be a scalar or a subrange type.

The Run Display is a stack which delimits the lexic levels within the run stack. The i-th entry points to the block mark for the last-invoked, currently active procedure at lexic level i.

The Local Variable Stack contains the storage for all variables and parameters of the procedures and functions. The variables of a PASCAL program are replaced by the compiler by (lexic level, order number) address couples which map into storage locations on this stack. The actual allocation of storage for the stack (and the mapping of the address couples into locations in this storage) is left to the code generation step.

The Local Variable Display is a stack which delimits the lexic levels within the local variable stack. The i-th entry locates those variables at lexic level i which are legally accessible.

The New Variable Stack contains the storage for variables allocated by the standard procedure new. The format of the entries in this stack is identical to that of the local variable stack.

The Expression Stack is used in the evaluation of expressions. Each entry on this stack can hold any scalar item. All arithmetic and relational operations take place using the top elements of this stack. This stack is also used to hold the parameters of the standard procedures and functions.

The Expression Display is a stack used to delimit the lexic levels within the expression stack. It is necessary in order to allow the clean-up of the expression stack if an exit is made from a function by the use of a goto statement. If in the middle of the evaluation of an expression, a function is invoked, and the invoked function exits by means of a goto, then it is possible that partially computed results may be present on the stack. This display is used to ensure that the valid portions of the stack are kept intact.

In many pseudo-machines, one finds the preceding three stacks merged. However, if one or all of these stacks are to be implemented to be independent of the others, (for example, on the IBM S/370, the expression stack could be implemented using the general purpose and floating point registers), then this separate definition of the stacks becomes helpful during code generation. Nevertheless, there is nothing preventing the implementation of this pseudo-machine on a real machine with two or three of the stacks combined, with a corresponding change in the displays. This definition appears flexible enough for many types of current hardware.
The Range Table contains the range information of all variables used in the program. Since the compilation step can perform all type checking necessary (parametric procedures and functions are not supported), only range information for the variables need be kept available for code generation. This table also contains the subscript information necessary to perform array indexing and record subfield accessing.

The range table, generated by the compile step and passed to the code generation step, has the following PASCAL definition:

```pascal
const Maximum_number_of_range_elements = 250;

type Range_index_type = 0 .. Maximum_number_of_range_elements;

Range_types = (Integer_type, Real_type, Char_type, Pointer_type, Array_type, Record_type, Set_type, File_type);

Storage_types = (Compacted, Not_compacted);

Range_element =
  record
    Storage: Storage_types;
    Subrange: boolean;
    Displacement: integer;
    case Variable_type: Range_types of
      File_type;
      Pointer_type: (Object_type: Range_index_type);
      Integer_type, Set_type, Char_type: (Lower_limit, Upper_limit: integer);
      Real_type: (Real_lower_limit, Real_upper_limit: real);
      Array_type: (Number_of_dimensions: integer);
      Record_type:
        (Number_of_subfields: integer)
  end;

Range = array [ 1 .. Maximum_number_of_range_elements ] of Range_element;

var Range_table: Range;
```

For all types, we record the storage type (compacted or not compacted), whether it is a subrange, its displacement if it is a subrange of a record, and its basic data type (integer, real, character, pointer, file, array, record, or set. For pointer types, we require information regarding its
object type to supply information for the code generator when processing references to the standard procedure new. Reals, integers and characters, if they are subrange types, as well as sets, also have their lower and upper limits recorded. For sets, these limits represent the lower and upper limits of the base elements. The number of dimensions is recorded for an array type, and the number of subfields (including tag fields and variants) is noted for record types.

The first four entries in the table are the four standard types:

1 integer
2 real
3 boolean
4 char

The fifth and sixth entries contain the description of the type text (packed file of char). The first six entries follow.

Range_table[1].Storage:=Not_compacted; { integer }
  .Subrange:=false;
  .Variable_type:=Integer_type;

Range_table[2].Storage:=Not_compacted; { real }
  .Subrange:=false;
  .Variable_type:=Real_type;

Range_table[3].Storage:=Not_compacted; { boolean }
  .Subrange:=true;
  .Variable_type:=Integer_type;
  .Lower_limit:=0;
  .Upper_limit:=1;

Range_table[4].Storage:=Not_compacted; { char }
  .Subrange:=false;
  .Variable_type:=Char_type;

Range_table[5].Storage:=Compacted; { text }
  .Subrange:=false;
  .Variable_type:=File_type;

Range_table[6].Storage:=Not_compacted;
  .Subrange:=false;
  .Variable_type:=Char_type;

Each type defined in the user's program has a range number associated with it by the compiler. This range number is actually the index into the range table for that type.

Entries are made into this table, for type definitions, as follows:

(a) For programmer types, (for example, type Colour=(Red, Orange, Yellow, . . . )), the range is recorded as an unpacked subrange of the integers with the lower limit
zero, and upper limit as the number of constants in the type minus one.

(b) For subrange types the lower and upper bounds are recorded and the containing type is either real, for a real subrange, character for character subranges, or integer for all other cases. Booleans are recorded as a subrange of the integers. This is the case because PASCAL allows the comparison of boolean values for both equality and inequality.

(c) File types are recorded as themselves. This is followed by an entry describing the component type of the file.

(d) Set types are recorded by noting the lower and upper limits of the base type.

(e) Pointer types are recorded by noting the range number of the object type. This information is necessary for references to the standard procedure new.

(f) Arrays are recorded by noting the number of dimensions. This is followed by as many entries as there are number of subscripts, the i-th entry denoting the type of the i-th subscript. These are then followed by an entry describing the element type of the array.

(g) Record types are recorded by noting the number of subfields. This count includes the number of fixed subfields, plus one for the tag field if present, plus the number of variants. This is followed by entries describing each subfield, tagfield, and variant.

The real machine storage requirements for each type can be computed at code generation time from the information contained in this table. The code generator, by maintaining a local variable stack which contains the attributes and range information of the variables, and an expression stack, which contains the attributes of the elements of expressions, at code generation time, can keep track of the displacements and lengths of the various variables used in the compiled program.

Finally, there are two registers which are part of this pseudo-machine. The Program Counter contains the address of the next-to-be-executed instruction in the Pseudo-Machine Code Table. The Statement Register contains the number of the currently executing statement. This information is used in the production of run time error messages.
The PASCAL Pseudo-Machine Language is presented in the following subsections. We will introduce the instructions, as required, in order to compile the various constructs and statements of PASCAL.

### 2.3.1 Locating and Fetching Variables and Constants

Variables are declared by the DECL instruction, which is discussed in section 2.3.3. For scalar variable access, two instructions are used:

- **PUSA (level, order)**: (PUSH Address) This instruction pushes the address of the variable specified by the address couple (level, order) onto the expression stack. The address couple (1,n) refers to the n-th variable declared at lexic level 1.

- **PUSV (level, order)**: (PUSH Value) This instruction pushes the value of the specified variable onto the expression stack.

In order to access an element of an array, the address of the array must first be pushed onto the expression stack. In a simple case, where the array is not a component of any other structured variable, this may be accomplished by PUSA (level, order). (More complex cases will be demonstrated by example later). Once the address of the array is on the expression stack, individual elements are accessed by first pushing the subscript values (after evaluation, if necessary) onto the expression stack, and then by use of one of the following instructions:

- **INDA n**: (INDex array Address) This instruction indexes the array whose address is the top element of the expression stack and stacks the address of the specified array element. The operand n specifies the number of subscripts on the expression stack that are to be used.
Both of these instructions pop the expression stack \( n+1 \) times before stacking the result.

In a manner similar to that of accessing an array element, subfields of records are accessed by first pushing the address of the record in which it is contained, onto the expression stack. The subfields are then accessed by use of one of the following instructions:

- **SFIA n** (Sub Field Address) The address of the \( n \)-th subfield of the record whose address is the top element of the expression stack is pushed onto the same stack.

- **SPIV n** (Sub Field Value) The value of the \( n \)-th subfield of the record whose address is the top element of the expression stack is pushed onto the same stack.

In both instructions, the address of the record is first popped off the expression stack, before the address or value of the subfield is pushed on. The instructions for accessing array elements and subfields or records are used recursively in accessing elements in records with arrays as subfields, and accessing subfields in arrays of records.

The accessing of constants requires its own set of instructions. Variants of the following instruction are used:

- **PCVx constant** (Push Constant's Value of type \( x \)) The value of the constant is pushed onto the expression stack.

In the above instruction, \( x \) must be one of I, R, S, C or B. These suffixes denote the type of the constant being accessed; they are defined as: I-integer, R-real, S-set, C-character constant of length one and B-boolean.

Use of the constant nil requires the following instruction:

- **PCVN** (Push Constant Value Nil) This instruction pushes the value of the constant nil onto the expression stack.

Accessing character strings of length greater than one makes use of:
PCAD constant (Push Constant's Address of type packed array of character) The address of the constant is pushed onto the expression stack.

Finally, one further instruction is required for the accessing of the values of variables. This is:

EVAL (EVALuate) This instruction replaces the address at the top of the expression stack by the contents of that location.

This is used in the accessing of scalar variables which are objects of pointers and are allocated by the standard procedure new. This instruction is also required for accessing call by reference parameters, since the address of the variable is passed.

2.3.2 Storing and Transfer of Values to Variables

There are two methods for storing results from the expression stack into variables. These make use of the following two instructions:

STOR (STORE) This stores the value located at the top of the expression stack into the location specified by the next to top element in the expression stack. These two elements are then popped off the stack. This method of value storing is used when an address calculation is necessary in order to access a location (for example, array elements).

STOW (level,order) (STOW) This instruction specifies that the value located at the top of the expression stack is to be stored into the location specified by the variable whose address couple is (level,order). The expression stack is then popped once. This is used when no address calculation is necessary in order to access the location (for example a scalar variable).
Also necessary is an instruction for storing the result of a function in the block mark associated with that function. Since only scalars may be the result of a function, the following instruction is sufficient:

**STPR level**

**STore Function Result** This instruction places the value located at the top of the expression stack into the block mark of the function currently executing, at the current lexical level.

Finally, the movement of structured data is facilitated by the use of the following instruction:

**MOV**

**MOVe Structured** This instruction copies the record or array specified by the top element of the expression stack to the location specified by the next to top element of this stack. Both addresses are popped off the stack when the operation is complete.
2.3.3 Function and Procedure Linkage

With every invocation of a function or procedure, an block mark is pushed onto the run stack, and the various displays are updated. Each block mark has the following format described section 2.2.

Note that this type of block mark and method of handling procedure invocation does not allow parametric procedures and functions. This is the case because this method does not record the environment of a procedure when it is passed as a parameter.

This block mark is pushed onto the run stack by the following instruction:

MKST n

(Mark run Stack) This instruction pushes a block mark onto the run stack. It sets both the dynamic and static links, saves the contents of the run, expression and local variable stack displays at level n, and the value of the statement register in this block mark.

After the block mark has been pushed onto the run stack, the parameters to be passed are evaluated and are pushed onto the appropriate stack. For user procedures and functions, the parameters are pushed onto the local variable stack. For standard procedures and functions, a block mark is not necessary and the parameters (which may only be scalars) are pushed onto the expression stack. We do not require a block mark for standard procedures since they are not recursive. In order to accomplish parameter passing for user procedures, the following instruction is required:

PARM n,kind

(PARaMeter) This instruction allocates sufficient space on the local variable stack to contain a variable of type n, where n is a range number. The kind operand specifies whether the parameter is call by value, or call by reference. In the latter instance, the space reserved on the local variable stack contains the address of the specified variable.

The parameters are now stored on the local variable stack in the manner described in section 2.3.2.

Once the parameters have been evaluated and placed onto the appropriate stack, the subprogram is called. Standard procedures are called using:
CSPR name  (Call Standard Procedure) The standard procedure "name" is called.

User procedures and functions are called using:

CUPR entry-point  (Call User Procedure) This instruction first places the return address in the block mark. It also interchanges the values of the save for the displays with the display entries at level n, the level of the called program, which is provided by the MKST instruction which precedes it. It then passes control to the user program at location entry-point.

Upon entry to the user program, the following instruction supplies the lexic level of the called procedure:

IXLV level  (Lexic Level) The following procedure will execute at lexic level "level".

The following instruction provides the necessary information for the code generator to reserve storage for each local variable, on the local variable stack:

DECL n,kind  (Declare) This instruction declares a variable whose range number is n. The operand n is a index into the range table. The operand kind may take on one of three values. It may specify that space is to be allocated for a locally declared variable. Or, it may specify that a variable is a call by value parameter, or a call by reference parameter. These latter two values act only to provide information to the code generator, but not to cause the allocation of space.

One instruction of this type is emitted for each variable and parameter. All declarations are collected together and placed at the head of the procedure to which they are local.

Upon completion of a user subprogram, control is returned to the invoking program by the use of the following instruction:

RETN type  (Return) This instruction returns control to the calling procedure, and restores the displays and statement register. When type=F
(the function return), the function result located in the block mark, is pushed onto the top of the expression stack. When type=P (the procedure return), no result is pushed onto the expression stack.
2.3.4 Expressions

2.3.4.1 Relational Operators

Expressions in PASCAL consist of two simple expressions surrounding a relational operator or a single simple expression. For the comparison of scalars, set and subrange types, the following instructions are used:

EQLx  
(EQual of type x) The top two elements of the expression stack are checked for equality. If the two elements are equal, the boolean true is pushed onto the expression stack after the two elements are popped off; otherwise, the boolean false is pushed on after the two elements are popped off.

NTElx  
(NoT Equal of type x) The top two elements of the expression stack are checked for inequality. The boolean result is pushed onto the stack after the two elements are popped off.

In the above two instructions, the letter x is replaced by one of the following to indicate the type of the elements on the expression stack: I-integer, R-real, C-one character, T-sets, B-Boolean, and P-pointers.

Comparison of structured types, such as arrays and records, may be done using the following instruction:

EQLS  
(EQual Structured) The structured variable whose address is the top element of the expression stack is compared to the structured variable whose address is the next to top element of the stack. Both addresses are popped from the stack and the result true is pushed onto the stack if the two variables were equal; the result is false otherwise.

NTELS  
(NoT Equal Structured) This instruction compares the structured variable whose address is the top element of the expression stack
against the structured variable whose address is the next to top element of the stack for inequality. If the two are unequal, the boolean true is stacked; otherwise false is stacked.

The following instructions implement the inequality operators, \(<\), \(\leq\), \(\geq\), \(>\), of PASCAL:

\begin{itemize}
  \item **LSTx** (Less Than of type \(x\)) This instruction compares the next to top element of the expression stack to the top element, and stacks the boolean true if the former is less than the latter; false is stacked otherwise.
  \item **GRTx** (Greater Than of type \(x\)) This instruction stacks true if the next to top element of the expression stack is greater than the top element; false is stacked otherwise.
  \item **LEQx** (Less than or Equal of type \(x\)) This instruction stacks true if the next to top element of the expression stack is less than or equal to the top element; false is stacked otherwise.
  \item **GEQx** (Greater than or Equal of type \(x\)) The boolean true is stacked if the next to top element of the expression stack is greater than or equal to the top element; otherwise, false is stacked.
\end{itemize}

The above four operations pop the top two elements off the expression stack before stacking the boolean result. The letter \(x\) in the above instructions represents the type of the operands on the expression stack, and is replaced by one of I-integer, R-real, C-single character, or E-boolean.

The only structured types on which relational operators other than equals and not equals may be used are packed arrays of characters. These types require their own set of instructions to implement the inequality operations:

\begin{itemize}
  \item **LSTS** (Less Than Structured) This instruction compares the packed array of characters whose address is the next to top element of the expression stack against the array whose address is the top element of the expression stack. The two
addresses are popped off the stack and the boolean result is pushed on; true is pushed on if the first array is less than the second array; otherwise, false is pushed on.

**GRTS**  
**G**reater **T**han **S**tructured) This instruction performs similarly to LSTS, except that the inequality checked for is greater than.

**LEQS**  
**L**ess than or **E**qual **S**tructured) This instruction operates similarly LSTS except that the inequality checked for is less than or equal.

**GEQS**  
**G**reater than or **E**qual **S**tructured) The inequality checked for is greater than or equal.

Several instructions are also included to handle set manipulations. These are:

**LINR**  
**L**eft set **I**ncluded in **R**ight) This instruction checks whether the next to top element of the expression stack is a set included in the top element of the stack. Both operands are popped from the stack before the boolean result is pushed on.

**RINL**  
**R**ight set **I**ncluded in **L**eft) The top element of the expression stack is checked for set inclusion in the next to top element of the stack. The boolean result is pushed onto the expression stack after the two operands are popped off.

**INNN n**  
**M**embership **IN** a **N** set) This instruction checks whether the next to top element of the expression stack is an element of the set (whose range number is n) which is the top element of the stack. Both operands are popped off the stack before the boolean result is pushed on.

2.3.4.2 Adding Operators
When used with set operands, the minus sign indicates set difference. The following instruction implements this use of it:

**DIFF**

(set DIFFerence) The top two elements of the expression stack are replaced by their set theoretic difference. The result is the set difference formed by taking the top element of the stack from the next to top element.

The language PASCAL also allows the construction of sets. This requires the following instruction:

**GNSS n**

(Generate Singleton Set) The top element of the expression stack is replaced by a set whose only element is this item. The operand `n` is a range number which supplies the information regarding the type of the base elements in the set, for the code generator.

The union of sets and the boolean "or", both represented by the same operator in PASCAL, have the following machine instructions:

**UNIN**

(UNION) The top two elements of the expression stack are replaced by their set theoretic union.

**LIOR**

(Logical Inclusive OR) The top two elements of the expression stack are replaced by their logical inclusive or.

### 2.3.4.3 Multiplying Operators

Terms in PASCAL are one or more factors connected by multiplying operators. The multiplying operators of PASCAL are multiplication, division, division with truncation, modulus, logical and and set intersection.

Multiplication is implemented by the following two instructions:

**MULI**

(MULtiply Integer) The top two elements of the expression stack are replaced by their product.
MULR  (MULtiply Real) The top two quantities on the expression stack which are both of type real, are replaced by their product.

Real division is implemented with the following instruction:

DIVR  (DIVide Real) The next to top element of the expression stack is divided by the top element. The result is pushed onto the stack after the two operands are popped off.

Integer division, (division with truncation), is implemented by the following instruction:

DIVI  (DIVide Integer) The next to top element of the expression stack is divided by the top element. Before the result is pushed onto the stack, the two operands are popped off.

The multiplying operator modulus has the following instruction:

MODD  (MODulus) The top two integer elements of the expression stack are replaced by the remainder after dividing the next to top element by the top element.

Finally, set intersection and logical "and", both represented by the same PASCAL operator, have the instructions:

INTR  (INTersection) The top two elements of the expression stack are replaced by their set theoretic intersection.

LAND  (Logical AND) The top two elements of the expression stack are replaced by their logical and.
Factors are the basic units in PASCAL expressions. These are variable identifiers, expressions enclosed in parentheses, array elements, record subfields, constants, function references and set expressions. The only operator applicable at this level is the logical "not". This operator is implemented by the instruction:

LMOT (Logical NOT) The boolean element at the top of the expression stack is replaced by its logical complement.
2.3.5 Statements

In the following sections, we present the sequence of pseudo-machine instructions generated for each statement of PASCAL.

2.3.5.1 Assignment Statements

With the instructions that have been introduced thus far, it is possible to emit code sequences for assignment statements. This can be best shown by a number of examples. The following fragment of a PASCAL program illustrates the pseudo-machine code generated for assignment statements. The address couples and subfield numbers appear as comments next to the variables which they represent. The code sequences appear as comments next to the statements.

type Matrix = array [ 1..10, 1..10 ] of integer;

Index = 1..2;

var A: array [ 1..5, 1..7 ] of boolean;   { (1,0) }
     B: array [ 1..5 ] of array [ 1..7 ] of boolean;
     { (1,1) }
C: boolean;   { (1,2) }
D: record   { (1,3) }
     E: integer;   { 0 }
     F: array [ 1..5 ] of integer;   { 1 }
     case G: Index of   { 2 }
           1: (H: integer);   { 3 }
           2: (Y: real)   { 4 }
end;
X: real;   { (1,4) }
K: integer;   { (1,5) }
L: array [ 1..10 ] of   { (1,6) }
     record
         M: integer;   { 0 }

-34-
N: real    { 1 }
end;

T: record    { (1,7) }
P: integer;    { 0 }
Q: array [ 1..10 ] of real    { 1 }
end;

R: ↑ Matrix;    { (1,8) }
S: array [ 1..7 ] of boolean;    { (1,9) }

begin
C:=A[2,3];    { PUSA (1,0)
                   PCVI 2
                   PCVI 3
                   INDV 2
                   STOW (1,2) }

C:=B[2][3];    { PUSA (1,1)
                   PCVI 2
                   INDA 1
                   PCVI 3
                   INDV 1
                   STOW (1,2) }

S:=B[2];    { PUSA (1,9)
                   PUSA (1,1)
                   PCVI 2
                   INDA 1
                   MOVS }

X:=D.H;    { PUSA (1,3)
                   SPIV 3
                   FLOT
                   STOW (1,4) }

K:=D.H;    { PUSA (1,3)
                   SPIV 3
                   STOW (1,5) }

K:=D.F[1];    { PUSA (1,3)
                   SPIA 1
                   PCVI 1
                   INDV 1
                   STOW (1,5) }

X:=L[3].N;    { PUSA (1,6)
                   PCVI 3
                   INDA 1
                   SPIV 1
                   STOW (1,4) }

-35-
2.3.5.2 Procedure Invocation

Procedure invocations have been discussed previously in subsection 2.3.3.

2.3.5.3 If Then Else Statements

This statement has two forms in PASCAL. It can be formed with and without the else clause. In the first instance, the statement form is:

IF <expression>
THEN <statement1>
ELSE <statement2>

The code sequence emitted for this would be:

```
   evaluate <expression>
   FJMP LAB1
   <statement1>
   UJMP LAB2
LAB1  <statement2>
LAB2  next <statement>
```

where LAB1 and LAB2 are PASCAL pseudo-machine addresses. For this statement, two new instructions have been introduced:

- **FJMP n** (False Jump) If the top of the expression stack contains the boolean false, then jump to location n; otherwise continue execution with the next statement. In any case, pop the expression stack once.

- **UJMP n** (Unconditional Jump) Control is unconditionally transferred to location n.

In the second form of the statement, the ELSE clause is omitted:

```
IF <expression>
    THEN <statement>
```

This results in the code sequence:

```
   evaluate <expression>
   FJMP LAB1
   <statement>
LAB1
```
2.3.5.4 Case Statement

The case statement is used to select a statement for execution from a number of choices based on the value of a so-called "tag" expression. It has the format:

```
CASE <expression> of
  <constant labels>: <statement1>;
  <constant labels>: <statement2>;
  
  <constant labels>: <statementn>
END
```

The code sequence emitted for this statement might be:

```
: evaluate <expression>
.
XJMP LAB1-m
LAB2
  . <statement1>
  .
UJMP LABn+2
LAB3
  . <statement2>
  .
UJMP LABn+2
LAB4
  .
    remaining <statement>'s
  .
UJMP LABn+2
LAB1 UJMP LAE3
UJMP LAE17
  .
  .
UJMP LAB22
LABn+2
```

The value of m in the above code sequence is the lower limit of the tag expression type. A new instruction has been introduced for this instruction:

```
XJMP n  \text{(indexed JUMP)}  \text{This instruction takes the value at the top of the expression stack, adds this to the address specified by the location n, and then jumps there. The expression stack is popped once.}
```

-38-
2.3.5.5 While Statement

The while statement requires no new pseudo-machine instructions for compilation. It has the format:

WHILE <expression> DO
  <statement>
END

The code sequence which results from the compilation of this statement is:

LAB1
  evaluate <expression>
  FJMP LAB2
  <statement>
  UJMP LAB1
LAB2

where LAB1 and LAB2 are pseudo-machine addresses.

2.3.5.6 Repeat Statement

The repeat statement also requires no new pseudo-machine instruction. It has the format:

REPEAT <statement list>
UNTIL <expression>

This statement produces the code sequence:

LAB
  <statement list>
  ...
  evaluate <expression>
  FJMP LAB

where LAB is a pseudo-machine address.
The for statement can be of two forms:

\[ \text{FOR} \ <\text{control variable}>\:=\langle\text{expression1}\rangle \ \text{TO} \ \langle\text{expression2}\rangle \ \text{DO} \ <\text{statement}> \]

or

\[ \text{FOR} \ <\text{control variable}>\:=\langle\text{expression1}\rangle \ \text{DCWNTO} \ \langle\text{expression2}\rangle \ \text{DO} \ <\text{statement}> \]

The following sequence of pseudo-machine code is emitted for the first form of this statement:

\[
\begin{align*}
\text{. evaluate } & \langle\text{expression1}\rangle \\
\text{. evaluate } & \langle\text{expression2}\rangle \\
\text{STOW } & \langle\text{dummy variable}\rangle \\
\text{STOW } & \langle\text{control variable}\rangle \\
\text{PUSV } & \langle\text{dummy variable}\rangle \\
\text{PUSV } & \langle\text{control variable}\rangle \\
\text{GEQI} & \\
\text{FJMP } & \text{LAB2} \\
\text{LAB1} & \langle\text{statement}\rangle \\
\text{PUSV } & \langle\text{dummy variable}\rangle \\
\text{PUSV } & \langle\text{control variable}\rangle \\
\text{NTEI} & \\
\text{FJMP } & \text{LAB2} \\
\text{PUSV } & \langle\text{control variable}\rangle \\
\text{INCR} & \\
\text{STOW } & \langle\text{control variable}\rangle \\
\text{UJMP } & \text{LAB1} \\
\text{LAB2}
\end{align*}
\]

In this code sequence, a new instruction has been introduced:

\[ \text{INCR} \] (INCRement) The top element of the expression stack is incremented by the value 1.

We also note the inclusion of the use of a dummy variable. The compiler declares this variable as a local variable in order to save the value of \langle expression2\rangle. The value of \langle expression2\rangle cannot be stored on the expression stack, because of the possibility that the loop may be exited using a GOTO statement. Hence, we require the dummy variable.
For the second form of the for statement, in which \textit{DOWNTO} replaces \textit{TO}, the preceding code sequence is produced with two changes: the GEQI instruction is replaced by a LEQI instruction, and the INCR instruction is replaced by a DECR instruction. These new instruction, \texttt{DECR}, operates as follows:

\begin{center}
\textbf{DECR} (DECRement) This decrements the top element of the expression stack by 1.
\end{center}

\subsection*{2.3.5.8 With Statement}

The \texttt{with} statement opens the scope of a record variable, and allows the access of subfields without the need for specifying the record name. It has the format:

\begin{verbatim}
WITH <record variable list>
DO <statement>
\end{verbatim}

Every time a subfield of a record in the \texttt{<record variable list>} is accessed, within the scope of the \texttt{WITH} statement, the base address of the record is first pushed on the expression stack, before evaluating the effective address. If the base address of the record requires computation of some sort (for example, pointer dereferencing), a new dummy variable, declared by the compiler, is used to save this base address. The reason for not using the display for storing the base address of the record(s) in the \texttt{<record variable list>}, is the same as that for the \texttt{FOR} statement -- the possibility of a \texttt{GOTO} statement occurring within the \texttt{WITH} statement, branching to a statement outside of the scope of the \texttt{WITH} statement.

\subsection*{2.3.5.9 Goto Statement}

The \texttt{goto} statement allows the transfer of the flow of control to another point in the program. This transfer can be made to one of:

(a) a location within the current scope level

(b) a location in an outer scope.
The former transfer can be implemented using the instruction:

UJMP location

The latter requires the emission of the instruction sequence:

CLEN level
UJMP location

A new instruction has been introduced to handle the second case of this statement:

CLEN level (CLEaNup) This instruction restores the displays to the lexic level specified as the operand.
2.3.6 Standard Functions and Procedures

2.3.6.1 Standard Functions

A number of standard functions are included in the language PASCAL. All of these have corresponding machine instructions in the PASCAL pseudo-machine. The following machine instructions operate on the top element of the expression stack, and replace it by the result of the application of the function:

**ABSI**  (ABSolute value of Integer) This instruction replaces the integer at the top of the expression stack by its absolute value.

**ABSR**  (ABSolute value of Real) The real number at the top of the expression stack is replaced by its absolute value.

**SQRI**  (SQuare Integer) The integer at the top of the expression stack is replaced by its square.

**SQRR**  (SQuare Real) The real number at the top of the expression stack is replaced by its square.

**ODDD**  (ODD) If the integer at the top of the expression stack is odd, it is replaced by the boolean true; otherwise, it is replaced by the boolean false.

**SUCC**  (SUCcessor) The top element of the expression stack is replaced by its successor. If no successor exists, the result is undefined.

**PRED**  (PREDecessor) The top element of the expression stack is replaced by its predecessor. If no predecessor exists, the result is undefined.

**ORDD**  (ORDinal) The character at the top of the expression stack is replaced by the ordinal number representing it in the character set.
CHRR (CHaRacter) The ordinal number at the top of the expression stack is replaced by the character which it represents in the character set.

EOFF (End Of File) The result is the boolean true if the specified file is in end of file status; otherwise, the result is false.

TRNC (TRuNCation) The top real element of the expression stack is replaced by an integer which is the integral part of the real number.

2.3.6.2 Standard Arithmetic Functions

Also included in PASCAL are a group of standard arithmetic functions. All are implemented by declared external procedures, and all operate in a manner similar to user defined procedures. This allows the user to define his own versions of these routines. All expect one parameter of type real and all return a real result. They are invoked by the CSPP instruction described in section 2.3.3.

SQRT (The square root function) The returned result is the square root of the parameter.

EXP (The exponential function) The returned result is the number e raised to the power of the value of the parameter.

LOG (The natural logarithm function) The result is the natural logarithm of the parameter.

SIN (The sine function) The result is the sine of the parameter.

COS (The cosine function) The result is the cosine of the parameter.

ARCTAN (The arctangent function) The returned result is the arctangent of the parameter.
A number of standard, built-in procedures are also included in PASCAL. These are invoked by the machine instruction:

```
CSPR name (Call Standard Procedure) This instruction is described in section 2.3.3.
```

Besides the standard arithmetic functions described in the previous section, this instruction may also be used to invoke the following procedures:

**PUT**
(Output to file procedure) This procedure appends the value of the buffer variable, whose address is the top element of the expression stack, to the file associated with it.

**GET**
(Input from file procedure) This procedure advances the current file position to the next component and assigns the value of this component to the buffer variable, whose address is the top element of the expression stack.

**OPENR**
(Reset file position procedure) This procedure resets the current file position of the file, whose address is the top element on the expression stack, to the beginning.

**OPENW**
(Delete file) This procedure discards the current value of the file variable, whose address is the top element of the expression stack.

**NEW**
(The new procedure) This procedure receives one parameter, a pointer to a type which is to be allocated, or two parameters, a pointer to a record type with variants, which is to be allocated plus a parameter which specifies the initial value of the first tag field occurring in the record. The code sequence emitted for a reference to new with one parameter is:

```
PUSA <pointer variable>
LENG n
```

-45-
For this procedure, we have introduced a new instruction:

LENG n  
LENGTH of type n  
This pushes the length of the type whose range number is n, onto the expression stack. This type is the object type of the pointer variable. The length of the type is determined in the code generation step.

For the two parameter call to new, the above sequence of instructions is extended to initialize the tag field:

PUSV <pointer variable>
SFIA n
PCVI <constant>
STOR

PACK  
(The pack procedure) This procedure copies an array whose address is the third from top element on the expression stack, to a packed array, whose address is the top element of the expression stack, for a length specified by the next to top element of this stack.

UNPACK  
(The unpack procedure) This procedure copies a packed array to an unpacked array. The source array address is the third from top element on the expression stack, the target array address is the next to top element and the length is the top element.

READ  
(Input and advance procedure) This procedure assigns the value of the buffer variable, associated with the standard input file, to the character parameter, whose address is the top element of the expression stack. The file is then advanced one component.
WRITE (Output and advance procedure) The value of the character parameter, which is the top element of the expression stack, is assigned to the buffer variable associated with the standard output file. The value of the buffer variable is then appended as the next component of the file.
Finally, there are three instructions in the PASCAL pseudo-machine which perform internal operations. These are:

**STSR n** *(Set Statement Register)* This instruction sets the statement register to the value n.

**CKRG n** *(Check Range)* This instruction does a range check on the top element of the expression stack, where n is the index into the range table. This instruction is emitted to perform range checking for subrange types and subscripts of arrays.

**STOP** *(STOP)* This instruction causes the immediate halt of execution. It is the last instruction to be emitted for any program.

In the preceding sections, we have presented a pseudo-machine for code generation and its application in compiling the various constructs of PASCAL. With the presentation of their semantics, we note that the pseudo-machine instructions may also be executed by an interpreter, as well as being used for code generation. A summary of these instructions appear in Appendix B.
In this chapter we discuss the first phase, the compiler, of a language translator system for PASCAL using a pseudo-machine for code generation. The following sections describe the components of the compiler.

3.1 General Overview

The compiler for the PASCAL language was written in the SUE System Language [Clark 1971]. A subset of the SUE Language, which has direct counterparts in the PASCAL Language, was used (see Appendix E). The SUE System Language was chosen because of its similarity to PASCAL (making the task of rewriting the compiler much simpler), and the availability of a compiler for it. The purpose of this was to allow the compiler to be rewritten, in the future, in PASCAL and have the compiler compile itself. This would result in a machine independent version of the PASCAL compiler.

The compiler uses the top-down, recursive descent technique of parsing. Each non-terminal of the language has a recursive procedure associated with it. The procedure associated with a non-terminal U is invoked by the syntax analyzer when it is searching for the U. Error recovery is accomplished by simply skipping the source text until either the start symbol or final symbol of the terminal associated with the non-terminal is encountered. The rules of syntax used in this parsing technique for PASCAL are summarized in the syntax charts in Appendix A.

As well as handling the parsing of the PASCAL source program, this section of the language translator system is also responsible for scanning, symbol table manipulation, and type checking. The output from this phase is the PASCAL pseudo-machine code, to which the original program has been transformed, plus a range table. This range table contains the information necessary for the code generator to determine the type and amount of storage required by each of the variables used in the program.
As was stated before, in order to generate a PASCAL compiler in a reasonable amount of time, as much of the Ammann-Jensen compiler interpreter system was used as possible. The overall structure of our PASCAL compiler is essentially the same as the Ammann-Jensen compiler. The scanner, method of parsing and type checking routines were left intact. The symbol table routines were altered to provide the information necessary for the PASCAL pseudo-machine described in Chapter 2. This was required because we changed the address computation scheme.

Much of the code emission routines suited our purposes, and their overall structure was adopted. However, changes were made to enable us to generate code for our more general method of addressing variables. This has allowed the implementation of this compiler to progress more quickly than would have been possible had we decided to develop our own routines.
3.2 Structure of the Compiler

3.2.1 The Symbol Table

The symbol table is organized as an unbalanced binary tree at each lexic level. There is a display stack, consisting of pointers, which acts as the root of each tree, the n-th entry being the root of the tree for the procedure at lexic level n. When a new scope level is entered, triggered by the appearance of a procedure/function definition or a WITH statement, a new entry is pushed onto the display. A binary tree is then created, with the root at the display, which contains the variables for that scope. When the scope is exited, the display is popped. Figure 3-1a illustrates a symbol table at lexic level 2, organized in this fashion.

Each binary tree depends on the order of the variable names which are entered. After the first variable is entered for a scope level at the root, succeeding entries are made at leaf nodes. If the name to be entered is lexically greater than a name at a node, the right link is chosen; otherwise the left link is chosen. The result of entering the variables whose names are J, B, Z, F, I, L, and C, in that order, is illustrated in Figure 3-1b.

Each node of the binary tree is an entry for one variable, type, constant or procedure/function name and contains the following information:

- We record the character representation of the name of the variable.
- There is a left and a right link, to which are attached later entries to the symbol table.
- A link points into the type stack (described in the next section), which provides the type information of this variable.
- A link is used to connect the subfields of a record, the names of the constants of a programmer type, or the parameter names to the procedure/function name.
- For manifest constants, the value of the constant is also recorded.
- For variables, the lexic level and order number are recorded, as well as information indicating
Figure 3-1a Symbol Table Using Unbalanced Binary Trees

Figure 3-1b Tree After Entries Have Been Made For J, B, Z, F, I, L, C
whether this variable is a call by value parameter, a locally declared variable, or a call by reference parameter.

- For subfields of a record, the field number is recorded.

- For procedures and functions, we make an entry indicating whether it is a standard procedure/function or a declared one. If the procedure/function is standard, the internal key number is noted. For a declared procedure/function the lexic level, address and type of procedure (parametric or actual) are recorded. If the procedure is not parametric, then two boolean values indicate whether the procedure/function is external (the standard arithmetic functions) and is declared forward. We allow procedure and function names and parameters to be defined before their bodies by the use of the forward construct. This permits recursive procedure invocations by brother procedures.
3.2.2 The Type Stack

The type stack is used to hold the type information of the variables in the symbol table. Each of its entries contains the following information:

- A boolean is used to prevent cycles when printing out the contents of the type table. These tables are printed out at the end of the compilation of a program.

- A pointer is used to index the range table (described in section 2.2).

- The type number of this type is recorded.

- For scalars, we record the type (standard or declared). If the scalar is declared, a link points back into the symbol table to its name.

- For subrange types, we record the containing type, and also the minimum and maximum values.

- For pointers, a link to the object type is recorded.

- For set types, a pointer to the set base type is recorded.

- For arrays, a pointer to the array element type and a pointer to the index type, as well as the number of elements are recorded.

- For records, a link points to the name of the first field in the symbol table. Another link points to the first variant (if any) which is recorded in the type stack.

- Files have a pointer to their component type recorded.

- For tagfields, we record a pointer to the name of the tagfield in the symbol table. Another pointer links to the first variant, in the type table, for which this type is the tagfield.

- Variants are noted by recording the value of their label. A pointer provides a link to the next variant, and another pointer links to any subvariant which may occur within this variant.
3.2.3 The Constant Stack

Integer, one character and boolean constants are recorded in the pseudo-machine instructions. The remaining constants are recorded in a constant stack. The entries in this stack hold the following information:

- we record the values of real constants and set constants.
- for string constants whose length is greater than one, we record the string and its length.

3.2.4 Exit Label List and Label List

Exit labels are recorded by noting the label value and a pointer to the next exit label. The exit label construct of the PASCAL language is not supported in this version of the PASCAL compiler. However, the syntax of the label declaration is checked.

The label list holds information pertaining to the local labels within a block. This information consists of the integer value of the label, the pseudo-machine address which it represents, a pointer to the next label entry and a boolean which indicates whether this label has been resolved. If the label has not been resolved, then the label address points to the last instruction which referred to this label. This instruction heads a list of instructions, chained together through the operand1 field, which referred to this label.

3.2.5 The Declaration and Pseudo-Machine Tables

For each procedure or function, we supply a declaration table and a pseudo-machine code table. These tables contain the declarations of the variables of the procedure and the code into which the PASCAL source program has been compiled, respectively. The declaration table consists of a series of range numbers of the variables in this procedure, along with a field which indicates whether this variable has been
declared locally, is a call by value parameter, or a call by reference parameter. The code table consists of a series of instructions whose format is

```
label     op-code     operand1,operand2
```

The label field may or may not be present, depending upon whether a branch is made to this instruction. Op-code is the numeric code of one of the PASCAL pseudo-machine instructions, and operand1 and operand2 specify the required operands of the instruction.
The compiler consists of a sequence of recursive modules which both parse and emit code for a PASCAL source program. Each module, when called, is responsible for locating the construct which it compiles and for recovering in case of error, to the point where the calling module may continue correctly.

The compiler begins by initializing the tables to contain the standard variables, types, constants, and procedures and functions. Processing of the PASCAL source program now begins at the block level.

Exit label declarations, if any are present, are the first items to be processed. The label value is checked to be an integer, and if valid, is entered into the exit list. Manifest constants are then processed. Their names are entered into the symbol table, and their values, if non-integer, are entered into the constant stack. Type definitions are processed and are entered into the symbol and type tables. It is at this point that entries are also made into the range table for these types. Next, the variable declarations are processed. Entries are made into the symbol table for all names, and entries are also made into the type and range tables if any new types are implicitly defined. PASCAL pseudo-machine instructions are emitted at this point for the allocation of the local variables. Finally, procedure and function definitions are handled. A new lexic level is entered for each, and the parameter definitions are processed. The rest of the procedure/function definition is handled by a recursive call to the block processing routine.

The body of the program is now processed. The body consists of a sequence of statements. All of these statements (except the assignment and procedure reference statements) begin with a reserved word. There is one routine for each type of statement. If the statement is preceded by a label, then this is inserted into the label list and any unresolved references to it are corrected.

Procedure function references are processed by a separate set of routines. A check is first made as to the type of the procedure/function reference. If the type is standard, then the linkage code for a standard procedure/function is generated. Otherwise, if the reference is to a declared procedure, code is generated which invokes the user defined procedure.

The expressions contained in the statements are processed by a series of recursive routines which handle expressions; simple expressions, which are the elements of expressions; terms, which are the elements of simple expressions; factors, which are the elements of terms; and
variables, which are the elements of factors. All of these routines emit PASCAL pseudo-machine instructions for evaluating the construct which they compile. Type checking is accomplished by maintaining record variables which contain the type of the last element in the expression, simple expression, term and factor to be processed.

The flow of control between these sections of the compiler is shown in Figure 3-2.
Figure 3-2 Flow of Control In Compiler
3.3 Summary

As the preceding sections indicate, the described PASCAL compiler will produce code and a range table from a PASCAL program, for our pseudo-machine described in Chapter 2. This code and table may be now be input into the second phase of a translator system using a pseudo-machine for code generation, called the code generator. This step can now produce from this information, machine code for a real machine.

The compiler has taken approximately five months to write and consists of approximately 8000 lines of SUE code. It is felt that a code generation step, written for the IBM S/370, would take about three to four months to complete by a person who is familiar with the described pseudo-machine and the IBM S/370.

And, now that the task of writing a compiler which converts PASCAL programs into the pseudo-machine code of the preceding chapter is completed, it need never be redone. This phase is machine independent. The only step which need be rewritten is the code generator, since this, of course, is machine dependent. We present an outline of the design of a code generator for the IBM S/370 in Appendix C.
A pseudo-machine for code generation has been developed. The code emitted by the compiler described in Chapter 3 is presently being verified.

In the following sections, we present the results of this investigation of using a pseudo-machine for code generation as an approach to translator writing.

### 4.1 Major Features of the PASCAL Pseudo-Machine for Code Generation

This investigation has provided the features necessary in a pseudo-machine that is to be used for code generation. For PASCAL, these include a range table describing the data types present in the program. This table is used by the code generator when processing variable declarations.

Another key feature of this pseudo-machine is the fact that no decisions regarding space allocation for variables are made in the compile step, but are left to the code generation step. This provides a great flexibility in allowing the pseudo-machine to be implemented on machines of different storage units (for example, word based machines and byte based machines). The method of addressing variables relies on the address couple, where \((l,n)\) refers to the \(n\)-th variable declared at lexic level \(l\). The mapping from order number to actual machine displacement is the task of the code generator.

Finally, another important feature of the pseudo-machine lies in the separation of the run, local variable, and expression stacks. This allows the code generator to allocate these stacks in the best location on the real machine for which it is generating code.
4.2 Effect of Using a Pseudo-Machine for Code Generation

4.2.1 On Machine Independence

This approach of using a pseudo-machine for code generation does indeed provide a machine independent means of writing a compiler. Because of its design (for example, no storage allocation decisions for variables, and separate stack locations), the resulting pseudo-machine is very flexible and adaptable to many types of hardware. Machine dependent operations, such as the allocation of storage for variables, and location of the stacks are performed at code generation time, when the characteristics of the host machine are known.

4.2.2 On Compiler Portability

As was stated in the previous chapter, the compiler was implemented using a subset of SUE which has counterparts in PASCAL. It is now an extremely simple task to recode the compiler in PASCAL. And once this has been done, the compiler coded in PASCAL could be compiled by the SUE version, producing pseudo-machine code and a range table which are both machine independent and portable. The only requirement now, for obtaining a running PASCAL compiler on any machine, is to write a code generator which converts the pseudo-machine code into the machine language of that machine.

In general, if any compiler, using a pseudo-machine for code generation, is capable of compiling itself, then one has a portable piece of software.

4.2.3 On Efficiency

-60-
Because the allocation of storage has been left to the code generator, we now have a means of using the storage units of any host machine, (for example, a byte on an IBM S/370, or a word on a PDP-10). This, then, allows a compaction of storage, and hence a saving over the conventional compiler-interpreter system.

The separation of the stacks is another important efficiency consideration. This allows the code generator to place the stacks in optimal locations on the host machine. For example, on an IBM S/370, the expression stack could be implemented using the general purpose and floating point registers.

Finally, real machine code executes more quickly than an interpreter can interpret intermediate code. Thus, if a program is to be used often, one need only save the output from the code generator. Peephole optimizers in the code generator could also improve the performance of its output.

However, one may incur a higher overhead in translating programs using this approach. It is quite possible that these two steps could require more time and space on some machines, than would a one step translator that directly produces the machine code of the host machine. However, we feel that the advantages of machine independence and portability gained by using this approach far outweigh this possible drawback.

Using this technique, then, one could implement relatively quickly, language translators for a high level language for several machines. The translators can produce real machine code competitive with code produced by language translators written specifically for those machines.

Thus, the overall effect of a pseudo-machine for code generation on machine independence, compiler portability and efficiency appears to be a highly desirable one.

This technique, then, of using a pseudo-machine for code generation appears to be a workable approach to translator writing. Enough flexibility is present in the pseudo-machine code to allow implementation on many types of hardware. Once the compiler phase of the system which generates the pseudo-machine code is written, it need not be redone. Only the code generator requires rewriting for each host machine.

It is hoped that the results of this investigation will aid future development of machine independent, portable and efficient software -- a vital issue in the world of computers today.
The text on this page appears to be written in a language that is not clearly identifiable. The text is dense and does not seem to follow a clear structure, making it difficult to extract meaning from it. It contains numerous words and phrases, some of which are repeated or seem to be out of context. Without more information or context, it is challenging to provide a coherent translation or interpretation.
REFERENCES


Dept. Of Computer Science, University of Toronto, 1971.

[Glass 1968]

[Goodfellow 1968]
Goodfellow, E. B., "Student Language Compiler for the IBM 1130 System", IBM Contributed Program Library, Form 1130-03.2.002, 1968.

[Grau 1962]

[Gries 1971]

[Hassitt et al 1973]

[Hoare and Wirth 1972]

[Jensen 1973a]

[Jensen 1973b]
Jensen, K., Correspondence to Holders of the PASCAL Compiler-Interpreter, Mar. 15, 1973.

[Kalmar 1973]

[Lee 1972]
Lee, E. S., (editor), "The Relation Between The Architecture of Computer Central Processors and Programming


Thurber and Myrna 1970]

[van der Poel 1962]

[Weber 1967]

[Welsh and Quinn 1972]

[Wilcox 1971]

[Wirth 1971]

[Wirth 1972]

[Wirth 1973]

[Wortman 1972]
Syntax Charts for PASCAL

The following pages contain the syntax charts for PASCAL. These are taken from the Revised Report On PASCAL [Wirth 1973].
PASCAL Pseudo-Machine Instructions

ABS1  ABSolute value Integer 43
ABSr  ABSolute value Real 43
ADDI  ADD Integer 30
ADDr  ADD Real 30
CHRR  Standard function chr 44
CKRG n  Check Range 48
CLEn n  CLEanup 42
CSPR name  Call Standard Procedure 25
CUPR entry point  Call User Procedure 25
DECL n,kind  DECLARE 25
DECR  DECREMENT 41
DIFF  set DIFFerence 31
DIVI  DIVide Integer 32
DIVr  DIVide Real 32
EOFF  Standard function eof 44
FQLB  EQUAL Boolean 27
FQLC  EQUAL Character 27
EQLI  EQUAL Integer 27
EQLP  EQUAL Pointer 27
EQLR  EQUAL Real 27
EQLS  EQUAL Structured 27
EQLT  EQUAL set 27
EVAL  EVALuate 22
FJMP location  False JUMP 37
FLOT  FLOaT 30
FLT2  FLoaT second 30
GEQB  Greater than or EQUAL Boolean 28
GEQC  Greater than or Equal Character 28
GEQI  Greater than or Equal Integer 28
GEQR  Greater than or Equal Real 28
GEQS  Greater than or Equal Structured 29

GNSS  n  Generate Singleton Set 31

GRTB  Greater Than Boolean 28
GRTC  Greater Than Character 28
GRTI  Greater Than Integer 28
GRTR  Greater Than Real 28
GRTS  Greater Than Structured 29

INCR  Increment 40

INDA  n  INDEX array Address 20
INDV  n  INDEX array Value 21

INNN  n  element INclusion 29

INTR  set INTERsection 32

LAND  Logical AND 32

LENG  n  LENGTH 46

LEQP  Less than or Equal Boolean 28
LEQC  Less than or Equal Character 28
LEQI  Less than or Equal Integer 28
LEQR  Less than or Equal Real 28
LEQS  Less than or Equal Structured 29

LIMR  Left set IN Right 29

LIOR  Logical Inclusive OR 31

LNOT  Logical NOT 33

LSTB  Less Than Boolean 28
LSTC  Less Than Character 28
LSTI  Less Than Integer 28
LSTR  Less Than Real 28
LSTS  Less Than Structured 28

LXLV  n  Lexic Level 25
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSST n</td>
<td>Mark Stack 24</td>
</tr>
<tr>
<td>MODD</td>
<td>MODulus 32</td>
</tr>
<tr>
<td>MOV3</td>
<td>MOVE Structured 23</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiply Integer 31</td>
</tr>
<tr>
<td>MULR</td>
<td>Multiply Real 32</td>
</tr>
<tr>
<td>NEGI</td>
<td>NEGate Integer 30</td>
</tr>
<tr>
<td>NEGR</td>
<td>NEGate Real 30</td>
</tr>
<tr>
<td>NTEB</td>
<td>Not Equal Boolean 27</td>
</tr>
<tr>
<td>NTEC</td>
<td>Not Equal Character 27</td>
</tr>
<tr>
<td>NTEI</td>
<td>Not Equal Integer 27</td>
</tr>
<tr>
<td>NTEP</td>
<td>Not Equal Pointer 27</td>
</tr>
<tr>
<td>NTER</td>
<td>Not Equal Real 27</td>
</tr>
<tr>
<td>NTES</td>
<td>Not Equal Structured 27</td>
</tr>
<tr>
<td>NTET</td>
<td>Not Equal Set 27</td>
</tr>
<tr>
<td>ODDD</td>
<td>Standard function odd 43</td>
</tr>
<tr>
<td>ORDD</td>
<td>Standard function ord 43</td>
</tr>
<tr>
<td>PARM n,kind</td>
<td>PARAMeter 24</td>
</tr>
<tr>
<td>PCAD constant</td>
<td>Push Constant Address 22</td>
</tr>
<tr>
<td>PCV3 constant</td>
<td>Push Constant Value Boolean 21</td>
</tr>
<tr>
<td>PCVC constant</td>
<td>Push Constant Value Character 21</td>
</tr>
<tr>
<td>PCVI constant</td>
<td>Push Constant Value Integer 21</td>
</tr>
<tr>
<td>PCV3N</td>
<td>Push Constant Value Nil 21</td>
</tr>
<tr>
<td>PCV3R constant</td>
<td>Push Constant Value Real 21</td>
</tr>
<tr>
<td>PCV3S constant</td>
<td>Push Constant Value Set 21</td>
</tr>
<tr>
<td>PRED</td>
<td>PREDecessor 43</td>
</tr>
<tr>
<td>PUSA (l,n)</td>
<td>Push Address 20</td>
</tr>
<tr>
<td>PUSV (l,n)</td>
<td>Push Value 20</td>
</tr>
<tr>
<td>RETN n</td>
<td>RETURN 25</td>
</tr>
<tr>
<td>RINL</td>
<td>Right set INcluded in Left 29</td>
</tr>
<tr>
<td>SPIA n</td>
<td>SubField Address 21</td>
</tr>
<tr>
<td>SPIV n</td>
<td>SubField Value 21</td>
</tr>
<tr>
<td>SQRI</td>
<td>Square Integer 43</td>
</tr>
<tr>
<td>SQR3R</td>
<td>Square Real 43</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>STFR n</td>
<td>Store Function Result 23</td>
</tr>
<tr>
<td>STOP</td>
<td>STOP 48</td>
</tr>
<tr>
<td>STOR</td>
<td>STORE 22</td>
</tr>
<tr>
<td>STOW (l,n)</td>
<td>STOW 22</td>
</tr>
<tr>
<td>STSR n</td>
<td>Set Statement Register 48</td>
</tr>
<tr>
<td>SUBI</td>
<td>Subtract Integer 30</td>
</tr>
<tr>
<td>SUPER</td>
<td>Subtract Real 30</td>
</tr>
<tr>
<td>SUCC</td>
<td>SUCCESSor 43</td>
</tr>
<tr>
<td>TRNC</td>
<td>TRuNCation 44</td>
</tr>
<tr>
<td>UJMP location</td>
<td>Unconditional Jump 37</td>
</tr>
<tr>
<td>UNIN</td>
<td>set UNION 31</td>
</tr>
<tr>
<td>XJMP location</td>
<td>indexed Jump 38</td>
</tr>
</tbody>
</table>
In this appendix, we present a brief outline of the use of the output of the compiler, described in Chapter 3, for the generation of real machine code for the IBM S/370. Many of the details are adapted from the implementation of the SUE System Language [Clark 1971]. No optimizations have attempted in this presentation.

**Storage For Data Types**

The following table provides the mapping from the data types held in the range table to the storage units on the IBM S/370.

<table>
<thead>
<tr>
<th>Range Table Type</th>
<th>S/370 Storage Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Fullword (32 bits)</td>
</tr>
<tr>
<td>Char</td>
<td>Byte (8 bits)</td>
</tr>
<tr>
<td>Real</td>
<td>Doubleword (64 bits)</td>
</tr>
<tr>
<td>Subrange 0 to 255</td>
<td>Byte (8 bits)</td>
</tr>
<tr>
<td></td>
<td>Halfword (16 bits)</td>
</tr>
<tr>
<td></td>
<td>Fullword (32 bits)</td>
</tr>
<tr>
<td>Subrange -32767 to 32767</td>
<td>Byte (8 bits)</td>
</tr>
<tr>
<td>Subrange -2^{31} to 2^{31}-1</td>
<td>Halfword (16 bits)</td>
</tr>
<tr>
<td></td>
<td>Fullword (32 bits)</td>
</tr>
<tr>
<td>Sets</td>
<td>Fullword (32 bits)</td>
</tr>
<tr>
<td>Pointers</td>
<td>Fullword (32 bits)</td>
</tr>
<tr>
<td>Files</td>
<td>DCE + Buffer</td>
</tr>
<tr>
<td>Call by Reference</td>
<td>Fullword (32 bits)</td>
</tr>
</tbody>
</table>
Before any code is generated for the S/370, the lengths of each data type in the range table are calculated in S/370 storage units, using the above table. Also, the relative locations of the subfields and variants within each record are calculated.

C.1 Location of Stacks and Displays

The expression stack is implemented using the general purpose registers starting from register 0, and working upwards, and all floating point registers. These will be referred to in this appendix as A0, A1, . . . , An, FA0, FA1, . . . . Since we are saving all registers at procedure entry, we do not require an expression display. The run stack and the local variable stack are merged into one stack, known simply as the run stack. The display for this run stack is located in the general purpose registers, starting at register 12 and working downwards. These will be referred to as D0, D1, . . . , Dn. This display removes the need for the static links in the block marks. The new variable stack grows towards the run stack, and its pointer is located in register 13. Registers 15 and 14 are used as base registers for the program control sections. Figure C-1 illustrates the layout.

Procedure Linkage

The procedure linkage mechanism presented here is a modified version of that found in [Clark 1971].
Figure C-1 Layout of Stacks And Display on S/370
But first, we present the format of the code at the head of each procedure (d2 is a displacement of an entry point address of a module declared local to this module):

```
Register 15 → Preliminary Code
  DELTA
  EP
  EP
  EP
  EP
  d2

OFFSET
```

The preliminary code checks for stack overflow. The DELTA field supplies the length of the area required for local variables and parameters on the run stack. The list of EP's is a list of the addresses of procedures nested within this one. These are collected and stored here by the code generator.

The format of the entries on the run stack are as follows. The first, the block mark, consists of four entries, each four bytes long.

<table>
<thead>
<tr>
<th>0</th>
<th>Return Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Dynamic Link</td>
</tr>
<tr>
<td>8</td>
<td>Return Address</td>
</tr>
<tr>
<td>12</td>
<td>Start Address of this Module</td>
</tr>
</tbody>
</table>

4 Fullwords (16 bytes)

Below this is a 64 byte area which acts as the register save area. (Note that the called program is responsible for saving and restoring any floating point registers it may use, in the
local variable area.)

<table>
<thead>
<tr>
<th>16</th>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 Fullwords (64 bytes)</td>
</tr>
</tbody>
</table>

Below this is a 4 byte field which is the statement register for this procedure:

<table>
<thead>
<tr>
<th>80</th>
<th>Statement Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Fullword (4 bytes)</td>
</tr>
</tbody>
</table>

Below this is the area for the local variables and parameters.

<table>
<thead>
<tr>
<th>84</th>
<th>DELTA</th>
</tr>
</thead>
</table>

We make the following assumptions at the time of the procedure linkage:

1) The call is from level n to level m.

2) The value of d1 must be such that Dn supplies addressability when registers and parameters are being stored. If this does not hold, we must compute d1(Dn) in a temporary register. (d1 = 84 + length of storage in bytes required for local variables and parameters at level n = 84 + DELTA(n)).
C.3 Calling Sequence

1. Save Registers and Set Up Block Mark

The MKST instruction generates the following code:

```
STM 0, 15, d1+16(Dn)  save registers
ST  Dn, d1+4(Dn)       set dynamic link
```

2. Obtain and Evaluate Parameters

The PARM instruction informs the code generator of a new local variable which actually belongs to the called procedure. However, it is accessed through an address couple which maps onto the stack location \(d1 + 84 + \text{length of previous parameters}\), above the contents of Dn.

3. Update Display

The CUPR instruction generates the following code plus that of 4 and 5.

```
LA  Dm, d1(Dn)     point to block mark
```
4. Load Register 15 With Entry Point Address

On uplevel and brother calls, only, we emit the following instruction to obtain the address of the called procedure:

\[ L \ 15,12(Dm-1) \quad \text{get base address of proc.} \]

The following instruction loads the entry point into register 15:

\[ L \ 15,d2(15) \quad \text{point to called procedure} \]

5. Branch and Link to Called Procedure

On procedure calls, the emitted instruction is:

\[ \text{BALR} \ 14,15 \]

6. Restore Registers on Return

The following code restores the registers for a procedure return:

\[ \text{LM} \ 0,15,16(Dm) \]

If a function was called, indicated by the following instruction in the pseudo-machine code not being \text{STSR} or \text{RETN}, then we emit the following instructions (\(Ai\) or \(PAi\) is the register which is the top free element of the expression stack):

-79-
The following code appears at the head of module:

```
LR T, Dm
A T, DELTA(15)
S T, STACKLIMIT
ENH OFFSET(15)
BAL 14, STACKOVERFLOW

DELTA stack length required
OFFSET STM 14, 15, 8(Dm)
XP 0, 0
ST 0, 80(Dm)
```

C.4 Entry Sequence

If register 14 was used for addressability, we must first restore it:

```
L 14, 8(Dm)
```

C.5 Return Sequence

The following instruction implements the pseudo-machine instruction RETN:

```
BR 14
```
C.6 Code Templates

We now present code templates for the remaining instructions in our PASCAL pseudo-machine. We shall use the following conventions:

1) $A_i$ is the accumulator which is the top free element of the expression stack; $A_{i-1}$ is the next to top element, etc.

2) even, odd is a free pair of general purpose registers used for division and multiplication

3) $T$ is a temporary register.
<table>
<thead>
<tr>
<th>Pseudo-Machine Instruction</th>
<th>S/370 Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSI</td>
<td>LPR Ai, Ai</td>
</tr>
<tr>
<td>ABSR</td>
<td>LPDR Ai, Ai</td>
</tr>
<tr>
<td>ADDI</td>
<td>AR Ai-1, Ai</td>
</tr>
<tr>
<td>ADDR</td>
<td>ADR Ai-1, Ai</td>
</tr>
<tr>
<td>CHRR</td>
<td>none required</td>
</tr>
<tr>
<td>CKRG</td>
<td>LR Ai+1, Ai</td>
</tr>
<tr>
<td></td>
<td>S Ai+1, =F'lower limit'</td>
</tr>
<tr>
<td></td>
<td>BNM *+2</td>
</tr>
<tr>
<td></td>
<td>BAL 14, RANGEERROR</td>
</tr>
<tr>
<td></td>
<td>L Ai+1, =F'upper limit'</td>
</tr>
<tr>
<td></td>
<td>SR Ai+1, Ai</td>
</tr>
<tr>
<td></td>
<td>BNM *+2</td>
</tr>
<tr>
<td></td>
<td>BAL 14, RANGEERROR</td>
</tr>
<tr>
<td>CLEN</td>
<td>not implemented</td>
</tr>
<tr>
<td>CSPR</td>
<td>dependent on standard proc.</td>
</tr>
<tr>
<td>CUPR</td>
<td>see sections C.3 and C.4</td>
</tr>
<tr>
<td>DECR</td>
<td>BCTR Ai, 0</td>
</tr>
<tr>
<td>DIFF</td>
<td>XR Ai, Ai-1</td>
</tr>
<tr>
<td></td>
<td>NR Ai-1, Ai</td>
</tr>
<tr>
<td>DIVI</td>
<td>LR even, Ai-1</td>
</tr>
<tr>
<td></td>
<td>SRDA even, 32</td>
</tr>
<tr>
<td></td>
<td>DR even, Ai</td>
</tr>
<tr>
<td></td>
<td>LR Ai-1, odd</td>
</tr>
<tr>
<td>DIVR</td>
<td>DDR Ai-1, Ai</td>
</tr>
<tr>
<td>EOFF</td>
<td>TM 16(Ai), X'40'</td>
</tr>
<tr>
<td></td>
<td>LA Ai, 1</td>
</tr>
<tr>
<td></td>
<td>BNZ *+2</td>
</tr>
<tr>
<td></td>
<td>XR Ai, Ai</td>
</tr>
<tr>
<td>EQLx x=E,C,I,P,T</td>
<td>CR Ai-1, Ai</td>
</tr>
<tr>
<td></td>
<td>LA Ai-1, 1</td>
</tr>
<tr>
<td></td>
<td>BE *+2</td>
</tr>
<tr>
<td></td>
<td>XR Ai-1, Ai-1</td>
</tr>
<tr>
<td>EQLR</td>
<td>CDR FAi-1, FAi</td>
</tr>
<tr>
<td></td>
<td>LA Ai-1, 1</td>
</tr>
</tbody>
</table>

-82-
<table>
<thead>
<tr>
<th>Pseudo-Machine Instruction</th>
<th>S/370 Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BE</strong> *+2</td>
<td></td>
</tr>
<tr>
<td><strong>XR</strong> Ai-1,Ai-1</td>
<td></td>
</tr>
<tr>
<td><strong>EQLS</strong></td>
<td><strong>CLC</strong> 0(length,Ai-1),0(Ai)</td>
</tr>
<tr>
<td></td>
<td><strong>LA</strong> Ai-1,1</td>
</tr>
<tr>
<td></td>
<td><strong>BE</strong> *+2</td>
</tr>
<tr>
<td></td>
<td><strong>XR</strong> Ai-1,Ai-1</td>
</tr>
<tr>
<td><strong>EVAL</strong></td>
<td><strong>L</strong> Ai,0(Ai)</td>
</tr>
<tr>
<td></td>
<td><strong>or</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LH</strong> Ai,0(Ai)</td>
</tr>
<tr>
<td></td>
<td><strong>or</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LR</strong> T,Ai</td>
</tr>
<tr>
<td></td>
<td><strong>XR</strong> Ai,Ai</td>
</tr>
<tr>
<td></td>
<td><strong>IC</strong> Ai,0(T)</td>
</tr>
<tr>
<td></td>
<td><strong>or</strong></td>
</tr>
<tr>
<td></td>
<td><strong>LD</strong> FAi,0(Ai)</td>
</tr>
<tr>
<td><strong>FJMP location</strong></td>
<td><strong>LTR</strong> Ai,Ai</td>
</tr>
<tr>
<td></td>
<td><strong>BZ</strong> location</td>
</tr>
<tr>
<td><strong>PLOT</strong></td>
<td>requires routine for</td>
</tr>
<tr>
<td></td>
<td>conversion</td>
</tr>
<tr>
<td><strong>FLT2</strong></td>
<td>requires routine for</td>
</tr>
<tr>
<td></td>
<td>conversion</td>
</tr>
<tr>
<td><strong>GEQx x=B,C,I</strong></td>
<td><strong>CR</strong> Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td><strong>LA</strong> Ai-1,1</td>
</tr>
<tr>
<td></td>
<td><strong>BNL</strong> *+2</td>
</tr>
<tr>
<td></td>
<td><strong>XR</strong> Ai-1,Ai-1</td>
</tr>
<tr>
<td><strong>GEQR</strong></td>
<td><strong>CDR</strong> FAi-1,FAi</td>
</tr>
<tr>
<td></td>
<td><strong>LA</strong> Ai-1,1</td>
</tr>
<tr>
<td></td>
<td><strong>BNL</strong> *+2</td>
</tr>
<tr>
<td></td>
<td><strong>XR</strong> Ai-1,Ai-1</td>
</tr>
<tr>
<td><strong>GEQS</strong></td>
<td><strong>CLC</strong> 0(length,Ai-1),0(Ai)</td>
</tr>
<tr>
<td></td>
<td><strong>LA</strong> Ai-1,1</td>
</tr>
<tr>
<td></td>
<td><strong>BNL</strong> *+2</td>
</tr>
<tr>
<td></td>
<td><strong>XR</strong> Ai-1,Ai-1</td>
</tr>
<tr>
<td><strong>GNSS</strong></td>
<td><strong>LR</strong> Ai+1,Ai</td>
</tr>
<tr>
<td></td>
<td><strong>LA</strong> Ai,1</td>
</tr>
<tr>
<td></td>
<td><strong>S</strong> Ai+1,=F'lower limit'</td>
</tr>
<tr>
<td></td>
<td><strong>SLL</strong> Ai,0(Ai+1)</td>
</tr>
<tr>
<td><strong>GRTx x=B,C,I</strong></td>
<td><strong>CR</strong> Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td><strong>LA</strong> Ai-1,1</td>
</tr>
<tr>
<td>Pseudo-Machine Instruction</td>
<td>S/370 Instruction Sequence</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>BH  <strong>+2</strong></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>XR  Ai-1,Ai-1</td>
<td></td>
</tr>
<tr>
<td>GRTG</td>
<td>CDR  FAi-1,FAi</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BH  <strong>+2</strong></td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>GRTS</td>
<td>CLC  0(length,Ai-1),0(Ai)</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BH  <strong>+2</strong></td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>INCR</td>
<td>A    Ai,=F'1'</td>
</tr>
<tr>
<td>INDA</td>
<td>(one dimensional arrays only)</td>
</tr>
<tr>
<td></td>
<td>LR  even,Ai</td>
</tr>
<tr>
<td></td>
<td>S    even,=F'lower limit'</td>
</tr>
<tr>
<td></td>
<td>SRDA even,32</td>
</tr>
<tr>
<td></td>
<td>M    even,=F'array element</td>
</tr>
<tr>
<td></td>
<td>length'</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,0(odd,Ai-1)</td>
</tr>
<tr>
<td>INDV</td>
<td>(one dimensional only)</td>
</tr>
<tr>
<td></td>
<td>LR  even,Ai</td>
</tr>
<tr>
<td></td>
<td>S    even,=F'lower limit'</td>
</tr>
<tr>
<td></td>
<td>SRDA even,32</td>
</tr>
<tr>
<td></td>
<td>M    even,=F'element length' and either</td>
</tr>
<tr>
<td></td>
<td>L    Ai-1,0(odd,Ai-1)</td>
</tr>
<tr>
<td></td>
<td>LH   Ai-1,0(odd,Ai-1)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>LR   T,Ai-1</td>
</tr>
<tr>
<td></td>
<td>XR   Ai-1,Ai-1</td>
</tr>
<tr>
<td></td>
<td>IC   Ai-1,0(odd,T)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>LD   FAi-1,0(odd,T)</td>
</tr>
<tr>
<td>INNN</td>
<td>LR   T,Ai-1</td>
</tr>
<tr>
<td></td>
<td>S    T=F'lower limit'</td>
</tr>
<tr>
<td></td>
<td>LA   Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>SLL  Ai-1,0(T)</td>
</tr>
<tr>
<td></td>
<td>NR   Ai,Ai-1</td>
</tr>
<tr>
<td></td>
<td>XR   Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>LA   Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>EZ   <strong>+2</strong></td>
</tr>
<tr>
<td></td>
<td>XR   Ai-1,Ai-1</td>
</tr>
<tr>
<td>Pseudo-Machine Instruction</td>
<td>S/370 Instruction Sequence</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>INTR</td>
<td>NR  Ai-1,Ai</td>
</tr>
<tr>
<td>LAND</td>
<td>NR  Ai-1,Ai</td>
</tr>
<tr>
<td>LENG</td>
<td>L   Ai,F'length'</td>
</tr>
<tr>
<td>LEQx  x=P,C,I</td>
<td>CR  Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BNH  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>LEQR</td>
<td>CDR  FAi-1,FAi</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BNH  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>LEQS</td>
<td>CLC  0(length,Ai-1),0(Ai)</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BNH  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>LINR</td>
<td>NR  Ai,Ai-1</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BZ  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>LIOR</td>
<td>OR  Ai-1,Ai</td>
</tr>
<tr>
<td>LNOT</td>
<td>LCR  Ai,Ai</td>
</tr>
<tr>
<td></td>
<td>A   Ai,F'1'</td>
</tr>
<tr>
<td>LSTx  x=P,C,I</td>
<td>CR  Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BL  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>LSTR</td>
<td>CDR  Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>BL  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>LSTS</td>
<td>CLC  0(length,Ai-1),0(Ai)</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BL  *+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td>MKST</td>
<td>see section C.3</td>
</tr>
<tr>
<td>Pseudo-Machine Instruction</td>
<td>S/370 Instruction Sequence</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>MODD</td>
<td>LR even, Ai-1</td>
</tr>
<tr>
<td></td>
<td>SRDA even, 32</td>
</tr>
<tr>
<td></td>
<td>DR even, Ai</td>
</tr>
<tr>
<td></td>
<td>LR Ai-1, even</td>
</tr>
<tr>
<td>MOVES</td>
<td>MVC 0 (length, Ai-1), 0 (Ai)</td>
</tr>
<tr>
<td>MULI</td>
<td>LR even, Ai-1</td>
</tr>
<tr>
<td></td>
<td>SRDA even, 32</td>
</tr>
<tr>
<td></td>
<td>MR even, Ai</td>
</tr>
<tr>
<td></td>
<td>LR Ai-1, odd</td>
</tr>
<tr>
<td>MULR</td>
<td>MDR Ai-1, Ai</td>
</tr>
<tr>
<td>NEGI</td>
<td>LCR Ai, Ai</td>
</tr>
<tr>
<td>NEGR</td>
<td>LCDR Ai, Ai</td>
</tr>
<tr>
<td>NTEX x=E, C, I, P, T</td>
<td>CR Ai-1, Ai</td>
</tr>
<tr>
<td></td>
<td>LA Ai-1, 1</td>
</tr>
<tr>
<td></td>
<td>BNE *+2</td>
</tr>
<tr>
<td></td>
<td>XR Ai-1, Ai-1</td>
</tr>
<tr>
<td>NTES</td>
<td>CDR FAi-1, FAi</td>
</tr>
<tr>
<td></td>
<td>LA Ai-1, 1</td>
</tr>
<tr>
<td></td>
<td>BNE *+2</td>
</tr>
<tr>
<td></td>
<td>XR Ai-1, Ai-1</td>
</tr>
<tr>
<td>ODDD</td>
<td>N Ai, =F'1'</td>
</tr>
<tr>
<td>ORDD</td>
<td>none required</td>
</tr>
<tr>
<td>PARM</td>
<td>see section C.3</td>
</tr>
<tr>
<td>PCAD</td>
<td>LA Ai, =C'...'</td>
</tr>
<tr>
<td>PCVx x=E, C, I, S</td>
<td>L Ai, =F'...'</td>
</tr>
<tr>
<td></td>
<td>L Ai, =X'FFFFFFFFFFFFF'</td>
</tr>
<tr>
<td>PCVR</td>
<td>LD Ai, =D'...'</td>
</tr>
<tr>
<td>PRED</td>
<td>BCTR Ai, 0</td>
</tr>
</tbody>
</table>

-86-
<table>
<thead>
<tr>
<th>Pseudo-Machine Instruction</th>
<th>S/370 Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUSA</strong></td>
<td>LA  Ai,displacement(,Dn)</td>
</tr>
<tr>
<td><strong>PUSV</strong></td>
<td>L   Ai,displacement(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,displacement(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,dispacement(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,Ai</td>
</tr>
<tr>
<td></td>
<td>or  Ai,dispacement(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  FAi,dispacement(,Dn)</td>
</tr>
<tr>
<td><strong>RETN</strong></td>
<td>see section C.5</td>
</tr>
<tr>
<td><strong>RINL</strong></td>
<td>NR  Ai-1,Ai</td>
</tr>
<tr>
<td></td>
<td>XR  Ai,Ai-1</td>
</tr>
<tr>
<td></td>
<td>LA  Ai-1,1</td>
</tr>
<tr>
<td></td>
<td>BZ  *=+2</td>
</tr>
<tr>
<td></td>
<td>XR  Ai-1,Ai-1</td>
</tr>
<tr>
<td><strong>SFIA</strong></td>
<td>LA  Ai,dispacement(,Ai)</td>
</tr>
<tr>
<td><strong>SFIV</strong></td>
<td>L   Ai,dispacement(,Ai)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,dispacement(,Ai)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,dispacement(,Ai)</td>
</tr>
<tr>
<td></td>
<td>or  T,Ai</td>
</tr>
<tr>
<td></td>
<td>XR  Ai,Ai</td>
</tr>
<tr>
<td></td>
<td>IC  Ai,dispacement(,T)</td>
</tr>
<tr>
<td></td>
<td>or  FAi,dispacement(,Ai)</td>
</tr>
<tr>
<td><strong>STFR</strong></td>
<td>ST  Ai,0(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,0(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,0(,Dn)</td>
</tr>
<tr>
<td></td>
<td>or  FAi,0(,Dn)</td>
</tr>
<tr>
<td><strong>STOP</strong></td>
<td>BAL  14,STOP</td>
</tr>
<tr>
<td><strong>STOR</strong></td>
<td>ST  Ai,0(,Ai-1)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,0(,Ai-1)</td>
</tr>
<tr>
<td></td>
<td>or  Ai,0(,Ai-1)</td>
</tr>
<tr>
<td></td>
<td>or  FAi,0(,Ai-1)</td>
</tr>
<tr>
<td>Pseudo-Machine Instruction</td>
<td>S/370 Instruction Sequence</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>STOW</td>
<td>ST  Ai,displacement(Dn)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>STH Ai,displacement(Dn)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>STC Ai,displacement(Dn)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>STD PAi,displacement(Dn)</td>
</tr>
<tr>
<td>STSP</td>
<td>MVI 2 (Dm),X'..'</td>
</tr>
<tr>
<td></td>
<td>MVI 3 (Dm),X'..'</td>
</tr>
<tr>
<td>SQRI</td>
<td>LR even,Ai</td>
</tr>
<tr>
<td></td>
<td>SRDA even,32</td>
</tr>
<tr>
<td></td>
<td>MR even,odd</td>
</tr>
<tr>
<td></td>
<td>LR Ai,odd</td>
</tr>
<tr>
<td>SQRR</td>
<td>MDR Ai,Ai</td>
</tr>
<tr>
<td>SUBI</td>
<td>SR Ai-1,Ai</td>
</tr>
<tr>
<td>SUBR</td>
<td>SDR Ai-1,Ai</td>
</tr>
<tr>
<td>SUCC</td>
<td>A Ai,=F'1'</td>
</tr>
<tr>
<td>TRNC</td>
<td>routine for conversion</td>
</tr>
<tr>
<td>UJMP location</td>
<td>B location</td>
</tr>
<tr>
<td>UNIN</td>
<td>OR Ai-1,Ai</td>
</tr>
<tr>
<td>XJMP location</td>
<td>B location(,Ai)</td>
</tr>
</tbody>
</table>
Unsupported Features of PASCAL

The following features of PASCAL are not supported by the compiler described in Chapter 3.

- parametric procedures
- exit labels and GOTO's leading out of scopes
- files

The following features of PASCAL are not supported by the code generator described in Appendix C.

- multi-dimensional arrays. This is not limiting since arrays of arrays are allowed.
- the packed attribute and the standard procedures pack and unpack.
APPENDIX E

Subset of SUE Utilized

Only certain features of the SUE System Language which have direct counterparts in PASCAL were used in writing the compiler described in Chapter 3. The major features are:

- The cycle exit end construct was limited to replacing the while do and repeat until constructs of PASCAL.
- Records and arrays were used.
- Macros were defined and stacks allocated to simulate the new standard procedure of PASCAL.
- Sets were not used, since the maximum number of elements they may contain is machine dependent. Instead, arrays of booleans were used.
- All character variable were limited to arrays of character(1).
- The do end control construct was used.
- The selector constructs, if then else end and case end were used.
Procedures and functions were utilized.