THE UNIVERSITY OF MANITOBA

A PASCAL COMPILER FOR IBM 360/370 COMPUTERS

by

W. BRUCE FOULKES

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

WINNIPEG, MANITOBA

October, 1975

THE UNIVERSITY OF MANITOBA LIBRARIES

"A PASCAL COMPILER FOR IBM 360/370 COMPUTERS"

by

W. BRUCE FOULKES

A dissertation submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

© 1975

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this dissertation, to the NATIONAL LIBRARY OF CANADA to microfilm this dissertation and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this dissertation.

The author reserves other publication rights, and neither the dissertation nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

To my Parents

ACKNOWLEDGEMENTS

I gratefully acknowledge the many comments and suggested improvements to the thesis made by members of the examining committee. Finally, I am indebted to my supervisor, Professor James M. Wells, for his encouragement and many valuable suggestions on all facets of the thesis.

ABSTRACT

This thesis describes a compiler to translate the programming language PASCAL into OS-compatible object modules for the IBM 360/370. The compiler employs an unusual design strategy and is rare among PASCAL compilers in that it is not a direct descendent of the original compiler developed for CDC machines. The compiler is intended for use in a production environment and therefore stresses the areas of fast compilation, compile-time error checking, run-time error detection and the production of efficient object code. Compilation strategy, internal organization and code generation are discussed in detail. Examples demonstrating the features and performance of the compiler are included.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis is divided into seven chapters. The current chapter presents a brief history of the development of the language PASCAL, and the original implementation for it. The objectives of the thesis are then indicated. Chapter 2 describes the related work in the area. Particular emphasis is given to the other implementations which have been attempted, and the burgeoning literature on the subject. Chapter 3 describes the implementation methods which were considered and the one which was adopted, and describes the results achieved in the light of similar attempts. Examples are given demonstrating compile-time and run-time error checks, and a large example is included which demonstrates many of the features of the language. Chapter 4 serves as a User's Manual and describes language constructs with reference to this implementation. Chapter 5 details the internal organization of the compiler, with particular emphasis on the design and use of the symbol table and other internal tables. The run-time organization and register usage are also described. Chapter 6 demonstrates the code generated for most constructs of the language. Run-time checking code and code optimization techniques are included in the examples. Chapter 7 presents conclusions on the project and outlines areas where further work is considered to be necessary.

The format of the thesis is intended to provide suitable break points for readers of varying degrees of interest. Reading to the end of Chapter 3 gives the historical background, an insight into the literature, a description of the approach used and the results achieved demonstrated with sample programs. If the reader is interested in using the compiler, continuing to the end of Chapter 4 provides the necessary information about language constructs. A reader who desires more detailed information about the internal organization of the compiler and the code which is generated should read to the end of Chapter 6.

1.1 HISTORICAL BACKGROUND

In recent years, George H. Richmond, of the University of Colorado has undertaken the task of editing the "Pascal Newsletter"; hereafter referred to as the Newsletter. To date, three Newsletters have been issued; No. 1 in January 1974 (Ric74a), No. 2 in May 1974 (Ric74b) and No. 3 in February 1975 (Ric75). The express purpose of the Newsletter is "to keep the PASCAL community informed about the efforts of individuals to implement PASCAL on different computers and to report extensions made to the language". In Newsletter No. 3, Richmond presents a history of PASCAL. The history of the development of PASCAL and the implementations on the original CDC machines is therefore presented with suitable excerpts from this source.

> "Pascal is an outgrowth of the early efforts of Dr. Niklaus Wirth to define a successor to Algol 60 by extension into the area of data definition facilities. In 1965, a proposal for such a successor was presented to an IFIP Working Group. It was

deemed not to be a sufficient advance over Algol 60 and was released for wider publication and appeared in the Communications of the Association for Computing Machinery (Wir66). Some of the aims of this proposal were to define a language that could be compiled quickly by a reliable translator, extend the utility of the language beyond strictly algebraic and numeric applications into areas such as information retrieval and symbol manipulation, and to oblige the programmer to express himself clearly without subverting the language to accomplish his task.

"These goals reappear in the original definition of the language Pascal (Wir71a). There was a desire to provide a language well suited to teaching programming as a systematic discipline based on fundamental concepts that were visibly reflected in the language. And Dr. Wirth wanted to demonstrate through an actual implementation that a significant and useful language can be realized in a reliable fast compiler.

"The first work on Pascal began in 1968 at Eidgenossische Technische Hochschule (ETH) in Zurich, Switzerland on a CDC 6000 system with the construction of a Pascal compiler written in Fortran (Wir71b). Although the compiler was completed, the result was an internal data structure and program contorted to fit the rules of Fortran. This compiler was unsuitable for translating at the source level into Pascal and a new compiler was started. It was entirely written in Pascal.

"Once the project had progressed to the point that the paper compiler could theoretically translate itself, the compiler was translated by hand into a low level language and an executable compiler was produced. This hand translation only took one man-month of effort. Admittedly, only enough of the language facilities to allow self-compilation had been implemented, but 60 percent of the final version was already there. Almost 3000 lines of code were produced before testing could begin. The full compiler was available in 1970 and this version is called PASCAL0.

"....The original report on Pascal provided a rigorous definition of the syntax of the language but only a verbal description of the semantics. The semantics were provided in 1972 by the publication of an axiomatic definition of Pascal (Hoa73). As part of this project, the syntax of the language was revised slightly and the revised report on Pascal was issued (Wir72a). To bring the existing compiler up to date, the necessary cosmetic changes, except for class variables which were retained from the previous version, were made and a new compiler was released. This version is refered to as PASCAL1 and was first available in late 1972. It is also known as PASCAL 6000-3.2.

> "However, the decision was made to rewrite the compiler completely. The changes included replacement of class variables with pointer variables to conform with the revised report, relocatable binary object code compatible with the standard CDC loader instead of a special absolute loader, introduction of a facility for separately compiled procedures and external Fortran subroutines, and new transput procedures for eliminating the need for a line control character and to accomodate CDC disk file structures. The new Pascal compiler was released in June 1974 and is called PASCAL2. It is also known as PASCAL 6000-3.4."

1.2 OBJECTIVES

The language PASCAL offers many advantages over the other languages in wide use. It has a simple, straightforward syntax which does not require complex and time-consuming parsing schemes. The statements offered by the language are easy to use and allow the writing of programs in a structured manner. The data types are especially appealing, allowing the definition of complex data structures, along with the introduction of scalar types and sets. The language is designed to be reasonably transportable between machines, and this is considered to be a great advantage.

The language was originally implemented on a CDC machine. For the language to gain wide acceptance, especially in North America, a compiler for IBM machines was a necessity.

It was therefore decided to write a PASCAL compiler for the IBM 360, which would not merely be an academic exercise, but would be usable in a production manner. It was decided to implement the language as defined in the Report (Wir71a, and later Wir72a), in the interests of portability. The temptation to introduce improvements

and modifications was therefore resisted in all but a few cases.

Once the decision was made to implement PASCAL, several considerations had to be taken into account before deciding on the implementation route.

It seemed that the principal users of PASCAL in the foreseeable future would be academic institutions. For a language and compiler to be used widely in a student environment, expecially at the undergraduate level, special conditions had to be met.

The compiler had to perform well when presented with source programs containing errors, and had to produce meaningful error messages. Errors in program logic, which do not become apparent until run time, had to be detected as soon as possible so that an indication of the problem could be given.

Another consideration in whether or not to use a language (and compiler) is the cost. This is principally a function of the memory size required by the compiler and compile speed. If the memory requirement is unusually high, this consideration alone can limit its use at some installations.

It was therefore decided to make the major design objectives the following:

1 - to perform exhaustive compile-time checks, taking full advantage of the redundancy of the language.

2 - to recover from syntax errors in the source as soon as possible so that meaningful error checking of the remainder of the source can be performed.

3 - to generate run-time checking code so that meaningful run-time error messages can be given, but at the same time allow this checking code to be suppressed if desired for increased efficiency in production runs.

4 - to make the compiler as fast as possible, and still meet the previous design objectives.

5 - to keep the memory requirements as low as possible, and still meet the other design objectives.

CHAPTER 2

RELATED WORK

2.1 IMPLEMENTATIONS

Since the original implementation of PASCAL on the CDC 6000 series, the language has been implemented on almost every major machine. The Newsletter reports that implementations have been accomplished on at least the following: CII IRIS 80, CII 10070, XDS SIGMA 7, DEC SYSTEM 10, IBM 360/370, UNIVAC 1108, ICL 1900 series and MULTIM. In addition projects are underway for: TI ASC, Data General 840, Raytheon 704, Univac AN/VYK-20, Honeywell G635, Burroughs 4700 and 6700 and the PDP-11/45.

The first transfer of PASCAL to another machine was accomplished by Drs. J. Welsh and C. Quinn of Quenn's University in Delfast, North Ireland, who transported PASCAL to the ICL 1900 Series Computers (Wel72). Their method was to change the code generation segment of the original compiler so that it produced ICL 1900 object code. A simulator for the ICL 1900 was written in PASCAL and run on the CDC 6000 at Zurich to test the resulting code. Careful preparation of the source programs plus an intensive short-term effort in the summer of 1971 resulted in the successful transfer of the compiler.

A compiler for IBM 360/370 computers has been produced at Grenoble, France. No documentation on this project seems to be available but it is believed that the technique used is the

following (Tas). The CDC compiler, written in PASCAL, was translated into PL/1, and the code generation portions were modified so that 360 Assembler is produced. A second compiler, written in PASCAL, was then compiled using the compiler written in PL/1, thus bootstrapping the language onto the IBM 360.

Mr. D. Russell and Mr. J. Sue have completed the bootstrap of a PASCAL compiler written in PASCAL on the IBM 360/370. The language implemented is a subset with just enough features to allow the compiler to compile itself. This implementation is based on the CDC 6600 compiler of January 1972 (Ric75).

Another method, besides the bootstrap technique, of transporting PASCAL to other machines, became available, and since has come into wide usage. This is the interpretive approach. Richmond in Newsletter No. 3 describes this development as follows:

> "During the past two years, another method for implementing PASCAL on computers other than the CDC 6000 series has been available from ETH. A portable Pascal compiler was released in early 1973 for this purpose. This first version is called PASCAL-P1. It was written to generate in an abstract language called P-code. By compiling the compiler, a P-code Pascal compiler is obtained. Thus, one only need provide an interpreter for P-code in order to obtain a Pascal compiler. However, the PASCAL-P1 compiler followed the original definition of Pascal (PASCAL0 compiler) and did not implement all the features available in the CDC 6000 version.
>
> "To remedy this problem, the portable Pascal compiler was rewritten and released in November 1974. It is called PASCAL-P2. The new compiler can be tailored to the target machine by specifying several parameters...."

This approach is being used by many people to accomplish a rapid implementation of PASCAL on other machines.

Mr. Alfred C. Hartmann and Robert S. Deverill, of the California Institute of Technology have implemented a subset of PASCAL using PASCAL-P1 (Dev74). At the present time, this system seems to be the one in widest use on IBM machines.

Other interpretive PASCAL projects for IBM machines have been reported by Dr. John Larmouth at Cambridge, England, and Dr. S. V. Rangaswamy at Bangalore, India (Ric75).

2.2 LITERATURE

Since PASCAL was originally defined (Wir71a), a considerable amount of literature has been written on the subject. Further language specifications followed: the Revised Report (Wir72), an axiomatic description of the language (Hoa73), and finally a user's manual (Jen74).

The Pascal Newsletter (Ric74a, Ric74b and Ric75) is playing an invaluable role in distributing information about further implementations of the language.

The language has been criticized by some people and praised by others. The most direct assault on the language was by A. N. Habermann (Hab73) who attempted to demonstrate that there are features in PASCAL which can cause problems if used in certain ways. This criticism was countered by Olivier Lecarme and Pierre Desjardins (Lec74b) who demonstrate that most of the problems indicated by Habermann were through poor programming practice, and state that it is possible to write poor programs in any language.

Many people consider PASCAL an ideal introductory language. Wirth uses PASCAL for his examples in his book on systematic programming (Wir73). Lecarme makes the case that PASCAL is an ideal language for teaching structured programming (Lec74a). This work includes an excellent bibliography with over a hundred references which serves as a very complete literature survey on these topics.

CHAPTER 3

IMPLEMENTATION STRATEGY

3.1 STRATEGIES CONSIDERED

As indicated in Chapter 2, several implementation strategies are possible. When work on this project was begun, the PASCAL-P approach was not available. It is doubtful that this approach would have been adopted in any event, as the resultant high core requirement and slow compile speed are inconsistent with the design objectives. Under the heading "PASCAL and Portability" in Newsletter No. 2 (Ric74b), Niklaus Wirth, the designer of the PASCAL language, states on this subject:

> "The Pascal-P approach is quite adequate and convenient, if efficiency of program execution is of no great significance. However, if the development of a high-quality compiler is the objective, a bootstrapping process on the basis of an interpretive Pascal-P system is very costly, and involves a large amount of reprogramming. It is clear that a different approach to the transportation of compilers themselves should be investigated."

The method of Welsh and Quinn (Wel72) was very attractive from the point of view of the time necessary to accomplish the transfer. The prime reason for not adopting this approach was the lack of availability of a CDC machine.

The method used at Grenoble (Tas) of translating the CDC PASCAL compiler into PL/1 was considered, but it was felt that the resulting compiler would not be as fast as it could be if it was written in a lower-level language.

The architecture of the CDC and IBM machines differs considerably and it was felt that any attempt to transport the compiler from the CDC to IBM machines, by either of the strategies discussed above, would lead to problems and in particular inefficiencies. If this approach was taken, the compile-time checking and error recovery would be dictated by that performed on the CDC compiler. In addition, modifying the code generators for another machine would not allow object code optimization to be accomplished as readily as if the compiler was designed from scratch for the target machine.

For these reasons it was decided to adopt an entirely different approach and write the compiler completely independently of the original PASCAL compiler for CDC machines.

3.2 STRATEGY ADOPTED

At the time the decision to write a PASCAL compiler was made, Rainer Kossmann was developing a Translator-Writing System called SYNTICS (Kos72), which he intended to use as a tool in the development of an ALGOL68 compiler. SYNTICS was the successor of an earlier system called RSYN (Kos72), which was PL/1 based. The RSYN system had been used by the author in the joint development, with Kossmann, of an interpreter for a logic language called DECLAB (Fou72).

SYNTICS is a PL360-based (Mal71 and Wir68) Translator-Writing System using a left to right, top-down parsing strategy. It accepts a modified BNF description of the syntax of a programming language, and optionally produces tables for a PL360 table-driven parser, or

generates an Assembler language parser. A scan routine is provided to read the source and present it to the parser in an acceptable form. Built-in phrases (BIPs) are pre-defined and allow the user to specify often used semantic actions, such as "look for an identifier", by simply using the name of the BIP. SYNTICS allows the user to specify points in the parse at which semantic actions are to take place. The names of procedures which are to perform the semantic actions (referred to as control phrases) are included in the desired places in the syntax description. These routines are then called when the indicated locations in the parse are reached. A facility for producing an analysis record of the parse is provided for multi-pass compilers. This analysis record consists of a tree-structured trace of the parse and can serve as the intermediate form between compiler passes. At present this feature is available only with the table-driven parser.

It was decided to use the SYNTICS system for the following reasons. The feature of being able to specify semantic actions at any desired place during the parse is very useful for early error detection during compilation. This allows an error message to be specified at precisely the point in the parse at which the error is detected.

The separation of syntax and semantics allows recovery for programs containing syntax errors to be handled almost entirely with syntax specifications. This means that changes can be made to the syntactic error recovery scheme without having to worry about possible side effects in the semantic routines.

Many trace features were provided, such as allowing a complete trace of the parse, which proved very useful especially during the design of the syntactic error recovery.

A compiler can be written in an extremely modular manner with different procedures called to handle the semantic actions at different places during the parse. This enables the semantic actions to take place at any time, to be well defined, and allows many semantic routines to be tested independently of other parts of the compiler.

Many of the BIPs and scan routine, etc. are common to any compiler and it seemed that much duplication of effort could be eliminated by using those routines which had already been developed.

Another major consideration was that the SYNTICS system was at a very early state of development and the possibility existed to influence the design to provide a more rounded system, which would have a wider applicability than if it was just designed to handle ALGOL68.

As Algol W (Wir66 and Bau68a) has many similarities to PASCAL, and was also implemented (Bau68b) using PL360, this compiler was investigated for common areas. Some ideas and code were actually adopted in the areas of code generation, arithmetic functions and input/output routines.

D. Moir wrote routines to allow formatted input/output in Algol W (Moi71), and some code was borrowed from this source.

3.2.1 FORM OF THE SYNTAX

The SYNTICS system requires that the syntactic description of the grammar be provided using the following notation. Terminals, nonterminals, BIPs and control phrases are all enclosed in angular brackets. Terminals are preceded by the symbol @ while BIPs are indicated with $ and control phrases are preceded by an *.

The main BIPs used in the PASCAL system are the following:

$IDENT - identifier
- a check is made to ensure that it is not a reserved word in the language.

$INT - integer

$DENOTATION - any of integer, real, string, TRUE, FALSE, hexadecimal constant etc.

$INTLABEL - integer treated as a character string.

When one of the above BIPs is specified, a system-provided semantic routine is called which looks in the source for the required construct. If it is found, the entry is hashed and SUCCESS is reported to the parser. If the construct is not found, the BIP reports FAILURE.

Control phrases also return either SUCCESS or FAILURE to the parser. Most control phrases just perform some operation, such as making an entry on the symbol table or generating code for some construct and therefore return SUCCESS when complete. A few routines are used to make decisions, such as whether or not an identifier is defined; these return SUCCESS or FAILURE depending on the test and thus have an effect on the direction taken by the parser.

Example:

```
- - <$IDENT> ( <*DEFINED> <OPTION1> | <OPTION2> )
```

The BIP $IDENT is called which searches in the source starting at the current pointer for an identifier. An identifier, as implemented in PASCAL, starts with a letter, followed by a sequence of letters, digits or underscores. If what appears to be an identifier is found, it is hashed to determine if it is in fact an identifier or a reserved word in the language. $IDENT returns SUCCESS only if the required token is found and is not a reserved word. The brackets after $IDENT indicate alternatives. The first alternative begins with a call on the control phrase DEFINED. This PL360 procedure uses the hash address provided by $IDENT to check if the identifier is currently defined. If it is, DEFINED reports SUCCESS and the parser proceeds to look for OPTION1. If DEFINED reports FAILURE the parser skips to the next alternative and looks for OPTION2.

Several other BIPs are used to control the syntactic error recovery. The recovery mechanism consists of a search for reserved words or other terminal symbols which mark distinctly recognizable points at which the parser can be reasonably sure of recovering. This parse recovery mechanism is highly recursive and the parser can get nested to many levels in the recovery part of the syntax. This mechanism was originally used on the SYNTICS system for lookahead when parsing ALGOL68 programs but proved useful for syntactic error recovery as well.

The following BIPs are used in the syntactic error recovery:

$SETLT - sets lookahead flag and reports SUCCESS.

$SETLF - sets lookahead flag and reports FAILURE.

$IFL - interrogates lookahead flag and reports SUCCESS if it is currently set; otherwise it reports FAILURE.

$IFNOTL - interrogates lookahead flag and reports SUCCESS if it is not currently set; otherwise it reports FAILURE.

Both $IFL and $IFNOTL reset the lookahead flag after interrogating it.

Example of Recovery Syntax:

```
                - - - ( <CONSTRUCT> | <ERROR<0,*>> );
<ERROR> := <$IFNOTL> <LOOKFOR>;
<LOOKFOR> := ( <@TERM1> | <@TERM2> | <@TERM3> ) <$SETLF>
           | <$IFL> <$SETLT>
           | <SCANTONEXTSYMBOL>;
```

The <0,*> means any number of occurrences including none. <ERROR> is looked for repeatedly until it returns FAILURE. <$SETLF> is used to set the lookahead flag and by reporting FAILURE causes the parser to backtrack the pointer before the terminal which was found. <$IFL> tests the lookahead flag. If set, <$SETLT> sets the flag again and reports SUCCESS for <ERROR>. <$IFNOTL> returns FAILURE on the next attempt ending the recovery sequence. If none of the terminals specified is found, <SCANTONEXTSYMBOL> skips the input pointer to the next terminal in the source and the process is repeated.

3.3 DETAILS OF PASCAL COMPILER

The PASCAL compiler is one-pass. Another pass would have made compile-time checking and code generation a little easier and might have decreased the core requirements, but would have had an adverse effect on compile speed, with the necessary production of an intermediate representation.

The table-driven parser was used during the early phases of compiler design. This provides many trace facilities and is inexpensive to compile. The generated Assembler parser takes a minute of CPU time to assemble and this is too costly to use when the syntax is changing often. These changes to the syntax were not in the form of modifications to the language, but merely changes to the description given to the SYNTICS system. Control phrases were inserted as the semantic routines were written and additional specifications were included to control the syntactic error recovery.

The Assembler parser is much faster than the equivalent table-driven parser and for this reason it was decided very early to use this parser in the final production version. The compiler with the Assembler parser is approximately 15K larger than, but runs at 3 times the speed of, a version with the table-driven parser. The fact that an analysis record is not available with the Assembler version of the parser was also a consideration in the decision to perform the analysis in one pass.

The syntactic error recovery mechanism used provides many levels of recovery from coarse to extremely fine. Its inclusion doubled the

length of the original syntax with a corresponding increase in the size of the final generated Assembler parser.

All of the semantic routines are written in PL360. These consist of procedures which have no expensive entry and exit code and therefore very little overhead. Registers are saved only when necessary, as several registers are always free for use by the semantic routines. In all, 194 control phrases (procedures) are specified in the syntax, while other routines are called only from these control phrases, giving a total of approximately 250 PL360 procedures for the PASCAL compiler, not counting those provided by the SYNTICS system.

The hash encoding scheme and symbol table mechanism (Fou73) described in detail in Chapter 5 were designed for use in the PASCAL compiler but now play an important role in the operation of SYNTICS. In addition, several of the BIPs described previously were added to SYNTICS because of needs which became apparent during the PASCAL effort.

The compiler performs exhaustive compile-time checks and produces approximately 130 different error and warning messages. Many errors are detected at compile time which although syntactically correct, would cause errors if allowed to proceed to execution.

Run-time checking code is optionally produced to check error-prone areas such as array subscripts, subrange assignments, values read into subranges or passed to them as parameters, values returned by SUCC and PRED, set assignments etc. These checks may be turned on or off on a line-by-line basis allowing checking code to be produced only for those routines which are undergoing testing and not impose this overhead on all routines included in the same compilation.

All run-time errors produce an error message detailing the problem, the location in the current segment, the invalid value if appropriate (i.e., in a case such as an array subscript out of bounds), and a traceback of all segments invoked.

Output from the compiler consists of standard IBM object modules. The format of the object module cards is given in (IBMb) while (Gri71) gives a description of their functions. Only IBM System 360 instructions (IBMc) are generated, allowing the compiler to be used on both IBM System 360's and 370's.

Linkage is not completely standard between PASCAL segments to increase the efficiency of accessing global variables, but appears to follow standard IBM conventions to any external segments, allowing linkage to routines written in other languages. It was considered to be very important that linkage to external routines be standard if PASCAL is to be used widely and not merely be a stand alone system.

The compiler, including all code and static data areas, is 112K bytes in length. The default table size in the version to be released is 40K bytes for a total length of 112 + 40 = 152K bytes. Due to system overhead, the compiler requires about 168K to run under VS2. The 112K of code and static data areas can be broken into 3 main divisions. The generated assembler parser is 24K while parser support tables are another 6K for a total of 30K bytes. SYNTICS system routines including the scanner, symbol table and block control routines, hash routine, error messages, BIPs and compiler initialization routines occupy 14K. This leaves 112 - 30 - 14 = 68K bytes for the main body of the PASCAL compiler and includes control phrases (semantic routines called

by the parser), and all routines for compile-time checking, code generation and table maintenance. The code generation is too heavily integrated with other routines to give a figure for this function alone. However, the register allocation routines and those involved in outputting the object modules require approximately 4K bytes.

The size of the compiler is larger than was originally planned but grew to its present size with modifications and improvements. Most large IBM installations have program regions of several hundred K bytes available and core is becoming less of a restriction with the passage of time. IBM's Fortran H compiler (IBMd) has been in use for many years and yet requires a region of at least 184K bytes. The size of the PASCAL compiler should only be a problem for very small installations.

All of the compiler tables are controlled through the use of displays (discussed in Chapter 5). This allows table sizes to be easily modified and makes checking for table overflows straightforward. The main table sizes may be modified by passing parameters on the EXEC card. The size limits of all tables, and all defaults for compiler flags (such as whether run-time checking code should be produced) are collected into an initialization routine called SETLIMIT, allowing the compiler to be readily modified to suit various needs.

At run time, the Input/Output routines and built-in functions are automatically included. A null program therefore requires a region of 14K bytes due to this overhead.

Compile speed for test programs has been in the range 125 - 200 lines per second on an IBM 370/158. This compares very favorably

with most production compilers which produce object modules. The set-up time for the compiler is approximately 0.4 sec.

The language PASCAL has changed several times since work on the project was begun. Initially, the original specification of the language (Wir71a) was used. Several constructs in the language changed with the Revised Report (Wir72). Hoare's axiomatic definition of the language (Hoa73) helped to define the semantics. Finally, the user's manual (Jen74) brought a few additional changes and helped to clarify some points. An attempt has been made to change with the revisions in the language to achieve compatibility with other implementations.

The only statement which can be made concerning the correctness of the compiler, is that it is written according to the author's interpretation of the semantics and its operation has been verified through running tests. There are certain to be some differences in the semantics between implementations in the areas which are not explicitly described but are left up to the implementor. The value of a FOR statement control variable after normal exit from the loop is not specified other than saying it is considered to be undefined. Some implementors (including this one) may make a decision as to the result of a logical expression without evaluating all of the operands and this might have an effect on the the overall logic due to side effects. There are several other vague areas of a similar nature where differences might occur, although this type of discrepancy should not affect most users.

There are some omissions and restrictions in this version of the compiler, and some extensions which are natural considering the

architecture of the target machine.

The omissions and restrictions are discussed in detail in Chapter 4 under the heading "LIMITATIONS IMPOSED BY THE COMPILER". The most important of these are the following:

- Only the standard input and output files SYSIN and SYSPRINT are supported. All I/O is accomplished through the use of READ, READLN, WRITE, WRITELN, EOLN and EOF. The program header is not used. SYSIN and SYSPRINT must be provided.
- Packed arrays and records are not supported.
- Only the simple forms of procedures NEW and DISPOSE are allowed. No garbage collection is performed.
- Branches to global labels are not supported.
- Subranges of characters are not allowed.
- The maximum nest allowed for procedure and function declarations is 5.
- All program segments are restricted to 4K bytes of code.

The major restriction in this version of the compiler is the limited file capability. The main reason for this omission was a lack of time. The compiler took 3 calendar years, working full-time during the summers and part-time during the winters for a total of approximately 24 man-months of work, to reach the present stage of development. It is estimated that an additional 6 to 12 man-months of work would be required to implement the complete file system.

The restricted file capability is considered to be adequate for most undergraduate needs, and the compiler can be used and evaluated while work on the files continues. Algol W (Wir66) also limits the

files to the standard Input and Output files and yet has enjoyed considerable usage. The areas in which this initial version of PASCAL is used are likely to be similar to those of Algol W.

Some extensions to the basic language are implemented, not with the aim of making the language supported incompatible with other PASCAL implementations, but because they are natural extensions for any implementation on an IBM machine:

- Three new standard scalar types are allowed. They are SHORT INTEGER, LONG REAL and STRING(n) where 1<=n<=256. These types are very natural and indeed have an equivalent in almost any language implemented on IBM machines.
- The Input/Output is not exactly standard. The I/O as described seemed too restrictive, especially in not allowing formatting on input. An attempt was therefore made to allow format specifications on input to parallel those of output as closely as possible. The result is that the I/O package is much closer to what people using IBM machines are accustomed than are the original specifications.

Built-in functions include: ABS, SQRT, EXP, LN, LOG, SIN, COS, ARCTAN, SQR, SUCC, PRED, ODD, ROUND, TRUNC, ORD, CHR, CARD and CPUTIME.

3.4 COMPARISONS WITH OTHER IMPLEMENTATIONS

Stastics on the features and performance of the PASCAL compiler have been presented. To put the results in perspective, a comparison is necessary with the other implementations which have been attempted for IBM machines. The approaches used by each were discussed in

Chapter 2. The main areas to be considered are the following:

- Whether or not the full language is supported, and if not, then what features have been omitted.
- The core requirements of the compiler or interpreter.
- The compile speed.
- Whether the projects are complete or in progress.

The PASCAL compiler / interpreter which seems to be the most widely used at the present time is the interpretive system developed at Caltech (Dev74). This system has been distributed to over 50 installations worldwide. It restricts the files to the standard input and output files. As well, formal parameter procedures and functions are not supported. Performance figures on an IBM 370/158 are available so that direct comparisons are possible. Figures on the core required by this system seem to vary widely, but the authors state that the compiler requires 270K bytes to compile itself. The compile speed is given as 13 lines of PASCAL source compiled per CPU-second. Hartmann (Har75) has stated that this system was used primarily for bootstrap purposes, and it is not intended to be maintained indefinitely.

A subset of PASCAL has been implemented at Stanford (Rus74, Gua75). The implementation is for the language described in the original definition of PASCAL (Wir71a). The major omissions are listed as the following (Rus74):

- "procedures and functions as formal parameters are not implemented"
- "real arithmetic is not completely implemented"
- "not all arithmetic functions (SIN, COS, etc.) are implemented"
- "packed records are not implemented"

In the introduction (Rus74), the following statement is made:

> "The current version is neither complete nor thoroughly tested. It is NOT suitable for normal use by students or other users. It should be viewed only as the first step in creating a usable implementation."

The following performance figures are available (Gua75). The compiler consists of 4700 lines of PASCAL source and compiles itself in 17 seconds on an IBM 370/168 in a region of 192 K bytes (the default at SLAC). The Stanford compiler is presently being distributed.

Although the Grenoble compiler project was underway when the present project was begun, little information about it has become available. A description has not been given in the Newsletter although the compiler is believed to be complete and distributed. At this time, no figures are available as to its core requirements or compile speed. A copy of the Grenoble compiler was obtained, but no effort was made to determine its characteristics because of the inadequate documentation which accompanied it.

A little information is available for the Cambridge interpreter (Lar75). The compile module compiles a null program in about 90K and a large one in about 120K at about 30 lines a second on an IBM 370/165. The run module runs a small program in about 40K. This system is presently being distributed.

In September 1974, Richmond mailed a questionaire to the people on the Newsletter mailing list. The answer to the question "Which version of Pascal are you using?" is very interesting.

The following is taken from Newsletter No. 3 (Ric75):

Version	% of Response	Number/Total Responses
Wirth, PASCAL2	41.6	47/113
Wirth, PASCAL1	25.7	29/113
Hartmann, IBM 370	9.7	11/113
Nagel, DEC 10	8.0	9/113
Wirth, PASCAL0	6.2	7/113
Welsh, ICL 1900	5.3	6/113
Thibault, CII IRIS	4.4	5/113
Amble, Univac 1100	3.5	4/113
Burger, CDC 6000	3.5	4/113

At the time of this survey it appears that the only version being used on IBM machines, was the interpreter developed at Caltech. This percentage figure is very low when compared to the percentage of machines which are IBM. It seems that although the Caltech system is available, it is too expensive to use on a large scale.

Brian Wichmann, of the National Physical Laboratory, Teddington, Middlesex, U.K. requested that Ackermann's Function be run on this PASCAL compiler. Ackermann's Function is a highly recursive function used to test the efficiency of calling procedures. This is the only PASCAL compiler for IBM machines for which figures are available. Table 1 gives the performance figures for all of the compilers tested on IBM machines. Brackets indicate that the figures are estimates.

TABLE 1

IBM COMPILERS

	Language/ Computer	Time per call (microseconds)	Instructions per call	Words per call
	Algol 60 360/75 F	870	(820)	(-)
	Algol 60 370/165 Delft	43.8	(142)	?
	Algol W 360/75	103	(97)	(16-45)
	Algol W 360/67 MK2	121	(74)	(16-45)
	PL/1 F 360/75 v4.3	270-550	(250-520)	(32-90)
	PL/1 F 360/65 v5.4	351	(212)	(70)
	P1/1 OPT 360/65 v1.2.2	101	(61)	68
*	PASCAL 370/158	39	42.5	30

From this it can be seen that the PASCAL compiler compares very favourably with Algol 60, Algol W, PL/1 F and PL/1 OPT compilers as far as efficiency of procedure calls is concerned.

Table 2 gives the performance figures for the PASCAL compilers which have been tested on various machines.

TABLE 2

PASCAL COMPILERS

Language/ Computer	Time per call (microseconds)	Instructions per call	Words per call
PASCAL CDC 6400	34	38.5	6
* PASCAL 370/158	39	42.5	30
PASCAL 1906S	19.1	32.5	11
PASCAL 1906A	31.5	32.5	11

It is much harder to draw conclusions from these figures because of the widely different architecture of the machines. The figures all seem to be approximately in the same range and do not vary as widely as those quoted in Table 1.

3.5 SAMPLE PROGRAMS

Three sample programs are included to demonstrate the workings of the compiler and the resultant code. Program 1/ demonstrates some of the error and warning messages produced by the compiler.

Program 2/ demonstrates some of the run-time errors which are detected by the run-time checking code. The same program is executed 6 times, each time with a different input value, resulting in a different run-time error. The run-time error messages are all shown on the page following.

Program 3/ comprises several procedures, each designed to demonstrate some particular feature of the language. The main program consists entirely of procedure calls. Each line of the main program produces a page of output. To increase readability, the output is included immediately following the procedure producing it, rather than at the end of the program.

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          UNIVERSITY OF MANITOBA 75.258 @ 19:14   PAGE   1

0  0000   00001  LABEL 10,20;
0  0034   00002  CONST PI = 3.14159;  N=10;
0  0034   00003  TYPE COLOUR = (RED,GREEN,BLUE,ORANGE,PINK,BROWN,BLACK);
0  0034   00004       DAY : (SUN,MON,TUES,WED,WED,THURS,FRI,SAT);
...WARNING(021)                 $                                          : INSTEAD OF =
***ERROR  (021)                                  $                         DUPLICATE IDENT
0  0034   00005       WKDAY = MON..FRI;
0  0034   00006       SUB=PINK..RED;
***ERROR  (015)                    $                                       MIN > MAX
0  0034   00007       SUB1=RED..FRI;
***ERROR  (016)                    $                                       CONST FROM DIFF SCALARS
0  0034   00008  VAR C:COLOUR;  D:DAY;  WD:WKDAY;
0  0034   00009      B:BOOLEAN;  S:STRING(10);  T:CHAR;
0  0034   00010      X:ARRAY(WKDAY,1..N) OF INTEGER;
0  0034   00011      I,J:INTEGER;  K:SHRTINT;  L:1..10;
0  0034   00012      T:INTEGER;  H_33:VECTOR;
***ERROR  (021)       $                                                     DUPLICATE IDENT
***ERROR  (020)                              $                             UNDEFINED IDENT
0  0034   00013      HUE:SET OF COLOUR;  Y:REAL;
0  0034   00014  FUNCTION TEST(VAR X,Y:REAL;  N:INTEGER):REAL;
1  0000   00015  VAR I:INTEGER; Z:REAL;
1  0066   00016     BEGIN
1  0066   00017     Z:=0.0;
1  006E   00018     FOR I:=1 TO N DO Z:=Z + X*Y;
1  00A6   00019     TEST:=Z
1  00A6   00020  END;
0  0034   00021  BEGIN
0  0034   00022     C:=SUN;  L:=-10;
***ERROR  (086)          $                                                  INV ASSIGN: DIFF SCALAR
***ERROR  (096)                    $                                       INV ASSIGN: OUT OF RANGE
0  0038   00023     WD:=SAT;  S:=10.314;
***ERROR  (096)          $                                                  INV ASSIGN: OUT OF RANGE
***ERROR  (088)                    $                                       INV ASSIGN: NOT CHAR OR STRING
0  003C   00024     HUE:=(.RED,BLUE,C,D,B,L);
***ERROR  (048)                    $                                       INV SET MEMBER
***ERROR  (048)                      $                                     INV SET MEMBER
***ERROR  (048)                        $                                   INV SET MEMBER
...WARNING(024)                          $                                 ) ASSUMED .)
0  0064   00025     I:=I J +?K;
>>>ERROR  (000)          $                                                  SYNTAX
***ERROR  (006)            $                                              INVALID SYMBOL
0  006C   00026     I:=(I+(I+J)*L));
>>>ERROR  (000)                    $                                       SYNTAX
0  0084   00027     I:=I+-J-+L;
>>>ERROR  (000)          $                                                  SYNTAX
>>>ERROR  (000)             $                                               SYNTAX
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)              UNIVERSITY OF MANITOBA 75.258 @ 19:14   PAGE   2

0  0094   00028     WHILE B BEGIN                                          ASSUMED DO
...WARNING(007)              $
0  009E   00029        HUE:=HUE + L;
***ERROR  (032)                  $                                      INV OPERAND WITH +
***ERROR  (089)                  $                                      INV ASSIGN: NOT SET OR DIFF SET
0  009E   00030        FOR I:=1 TO 10 DO BEGIN
0  00C2   00031           X(SAT,-3):=X(MON,4);
***ERROR  (073)               $                                         ARRAY SUBSCRIPT OUT OF RANGE
***ERROR  (073)                  $                                      ARRAY SUBSCRIPT OUT OF RANGE
0  00DA   00032           GO TO 20;
0  00DE   00033         10: Y:=TEST(3.7E-2,Y+PI,N);
***ERROR  (074)                        $                                ACTUAL PARM MUST BE A VARIABLE
***ERROR  (074)                           $                             ACTUAL PARM MUST BE A VARIABLE
0  011C   00034           I:=I+4;
***ERROR  (081)               $                                         MODIFIED CONTROL VAR
0  0128   00035        END;
0  012C   00036        GO TO 10;
***ERROR  (079)               $                                         BRANCH INTO STRUCTURED STAT
0  0130   00037     END;
0  0134   00038     CASE WD OF
0  0134   00039        SAT: Y:=SIN(Y);
***ERROR  (055)            $                                            INV CASE LABEL
0  0166   00040        WED: Y:=COS(Y);
0  017A   00041        RED: Y:=SQRT(Y);
***ERROR  (055)            $                                            INV CASE LABEL
0  018E   00042     END;
...WARNING(018)     $                                                   ; BEFORE END
0  0198   00043     I@:=J;  B(I,J):=10;  I:=X.A;
***ERROR  (071)       $                                                 NOT POINTER-@?
***ERROR  (029)            $                                            NOT ARRAY-SUBSCRIPT?
***ERROR  (087)                  $                                      INV ASSIGN: NOT BOOLEAN
***ERROR  (067)                     $                                   NOT RECORD-FIELD?
***ERROR  (095)                     $                                   INV ASSIGN: INCOMPATIBLE TYPES
0  01A0   00044    10: IF I = 10 THEN Y:=SQRT(Y); ELSE Y:=SQR(Y);;
***ERROR  (059)        $                                                DUP LABEL DEF
...WARNING(017)                                    $                    ; BEFORE ELSE
0  01CA   00045  END;
***ERROR  (061)     $                                                   MISSING LABEL: 20
...WARNING(013)     $                                                   ASSUMED .
COMPILE TIME:     0.262 SECOND(S)
PARSED SUCCESSFULLY
        6 WARNING(S) DETECTED
       34 ERROR(S) DETECTED
RETURN CODE 0012
```

Program 2/

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)                    UNIVERSITY OF MANITOBA

0  0000    00001   /* PROGRAM TO DEMONSTRATE RUN-TIME ERRORS                                  */
0  0000    00002   /*      THE PROGRAM IS RUN 6 TIMES WITH THE FOLLOWING INPUT CARDS:         */
0  0000    00003   /*      1                                                                  */
0  0000    00004   /*      2                                                                  */
0  0000    00005   /*      3                                                                  */
0  0000    00006   /*      4                                                                  */
0  0000    00007   /*      JOHN BROWN                                                         */
0  0000    00008   /*      -1                                                                 */
0  0000    00009
0  0000    00010   TYPE COLOUR = (RED,GREEN,BLUE,YELLOW,PINK,BLACK);
0  0034    00011   VAR I,J : INTEGER;
0  0034    00012       M : COLOUR;
0  0034    00013       S : SET OF 1..10;
0  0034    00014       A : -10..20;
0  0034    00015       X : ARRAY(1..10) OF INTEGER;
0  0034    00016
0  0034    00017   FUNCTION SUM(J:INTEGER):INTEGER;
1  0000    00018   VAR X:INTEGER;
1  004E    00019   BEGIN
1  004E    00020      X:=0;
1  0054    00021      FOR A := 1 TO J DO X:=X + A;
1  0098    00022      SUM:=X
1  0098    00023   END;
0  0034    00024
0  0034    00025   BEGIN
0  0034    00026      WRITELN('1':1);
0  0046    00027      J:=23;  M:=BLACK;
0  0056    00028      S:= (.1..3.);
0  006A    00029      READLN(I);
0  007C    00030      WRITELN('INDEX=',I);
0  00A0    00031      CASE I OF
0  00A0    00032         1: X(J):= X(J) + 1;
0  010A    00033         2: M:= SUCC(M);
0  0126    00034         3: S:=S + (.9,J.);
0  0158    00035         4: I:=SUM(J)
0  0160    00036      END;
0  018C    00037   END.
COMPILE TIME:     0.133 SECOND(S)
PARSED SUCCESSFULLY
       NO WARNING(S) DETECTED
       NO ERROR(S) DETECTED
```

```
 INDEX=           1
*** RUN ERROR: ARRAY SUBSCRIPT OUT OF RANGE                    AT OFFSET 00CE IN SEGMENT PASCAL      VALUE=          23
RETURN CODE 0016

 INDEX=           2
*** RUN ERROR: SCALAR VALUE UNDEFINED(RESULT OF FUNCTION SUCC) AT OFFSET 011E IN SEGMENT PASCAL      VALUE=           6
RETURN CODE 0016

 INDEX=           3
*** RUN ERROR: INVALID SET MEMBER                              AT OFFSET 0150 IN SEGMENT PASCAL
RETURN CODE 0016

 INDEX=           4
*** RUN ERROR: FOR STATEMENT LIMIT OUT OF RANGE                AT OFFSET 0060 IN SEGMENT SUM         VALUE=          23
         EXECUTING AT OFFSET 0172  IN SEGMENT PASCAL
RETURN CODE 0016

*** RUN ERROR: NUMERIC INPUT - INVALID CHARACTER               AT OFFSET 0074 IN SEGMENT PASCAL
INPUT RECORD: -->   JOHN BROWN                                                  <--
                    |
RETURN CODE 0016

 INDEX=          -1
*** RUN ERROR: CASE INDEX NEGATIVE                             AT OFFSET 00B6 IN SEGMENT PASCAL      VALUE=          -1
RETURN CODE 0016
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          UNIVERSITY OF MANITOBA

0  0000   00001  TYPE
0  0034   00002     COLOUR=(WHITE,YELLOW,ORANGE,PINK,RED,BLUE,PUPPLE,BROWN,BLACK);
0  0034   00003     WEEK=(SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY);
0  0034   00004     DATE=RECORD
0  0034   00005           MO:(JAN,FEB,MAR,APR,MAY,JUNE,JULY,AUG,SEPT,OCT,NOV,DEC);
0  0034   00006           DAY:1..31;
0  0034   00007           YEAR:INTEGER
0  0034   00008        END; /* DATE */
0  0034   00009     NAME=STRING(30);
0  0034   00010     STATUS=(MARRIED,WIDOWED,DIVORCED,SINGLE);
0  0034   00011     VECTOR=ARRAY(1..100) OF REAL;
0  0034   00012     PERSON=RECORD
0  0034   00013           N:RECORD
0  0034   00014              FIRST,LAST:NAME
0  0034   00015             END;
0  0034   00016           SINNBR:INTEGER;
0  0034   00017           SEX:(MALE,FEMALE);
0  0034   00018           BIRTH:DATE;
0  0034   00019           DEPENDENTS:INTEGER;
0  0034   00020           CASE MS:STATUS OF
0  0034   00021              MARRIED,WIDOWED:(MDATE:DATE);
0  0034   00022              DIVORCED:(DDATE:DATE;
0  0034   00023                        FIRSTD:BOOLEAN);
0  0034   00024              SINGLE:(INDEPENDENTRES:BOOLEAN)
0  0034   00025            END; /* PERSON */
0  0034   00026
0  0034   00027   /*-----------------------------------------------------------------*/
0  0034   00028   /* MAXIMUM VALUE IN FIRST N ELEMENTS OF VECTOR A */
0  0034   00029  FUNCTION MAX (VAR A:VECTOR;  N:INTEGER): REAL;
1  0000   00030     VAR I:INTEGER;
1  0056   00031     X:REAL;
1  0056   00032  BEGIN
1  0056   00033     X:=A(1);
1  0062   00034     FOR I:=2 TO N DO
1  0086   00035        IF X < A(I) THEN X:=A(I);
1  00D6   00036     MAX:=X
1  00DA   00037  END; /* MAX */
0  0034   00038
0  0034   00039   /*-----------------------------------------------------------------*/
0  0034   00040   /* COMPUTE N FACTORIAL   */
0  0034   00041  FUNCTION FACT(N:INTEGER): INTEGER;
1  0000   00042  BEGIN
1  004E   00043     IF (N = 0) | (N = 1) THEN FACT:=1 ELSE FACT:=N*FACT(N-1)
1  0086   00044  END; /* FACT */
0  0034   00045
0  0034   00046   /*-----------------------------------------------------------------*/
0  0034   00047   /* GREATEST COMMON DIVISOR OF M AND N */
0  0034   00048  FUNCTION GCD(M,N:INTEGER): INTEGER;
1  0000   00049  BEGIN
1  005A   00050     IF N = 0 THEN GCD:=M ELSE GCD:=GCD(N,M MOD N)
1  0092   00051  END; /* GCD */
```

MANITOBA PASCAL VERSION 1 (JUNE 1975) TEST OF RECURSIVE FUNCTION CALLS

```
0  0034   00052  PROCEDURE TEST1;  /* CALLS RECURSIVE FUNCTION FACT  */
1  0000   00053  CONST N=10;
1  003C   00054  VAR I:INTEGER;
1  003C   00055  BEGIN
1  003C   00056     FOR I:=0 TO N DO WRITELN(I,' FACTORIAL=',FACT(I));
1  0072   00057  END;  /*  TEST1  */
0  0034   00058
0  0034   00059   /*-------------------------------------------------------------------*/
0  0034   00060  PROCEDURE TEST2;
1  0000   00061  VAR I,J:INTEGER;
1  003C   00062  BEGIN
1  003C   00063     WRITELN(5,25,' GCD=',GCD(5,25));
1  00A6   00064     WRITELN(36,84,' GCD=',GCD(36,84));
1  0110   00065     WRITELN(99,33,' GCD=',GCD(99,33));
1  017A   00066     WRITELN(0,10,' GCD=',GCD(0,10));
1  019E   00067  END;  /*  TEST2  */
```

```
TEST1
         0   FACTORIAL=           1
         1   FACTORIAL=           1
         2   FACTORIAL=           2
         3   FACTORIAL=           6
         4   FACTORIAL=          24
         5   FACTORIAL=         120
         6   FACTORIAL=         720
         7   FACTORIAL=        5040
         8   FACTORIAL=       40320
         9   FACTORIAL=      362880
        10   FACTORIAL=     3628800

TEST2
         5          25   GCD=           5
        36          84   GCD=          12
        99          33   GCD=          33
         0          10   GCD=          10
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          GENERATE RANDOM NUMBERS AND FIND LARGEST

0  0034   00068  PROCEDURE TEST3;
1  0000   00069  CONST N=25;
1  003C   00070  VAR I,X:INTEGER;
1  003C   00071     A:VECTOR;
1  003C   00072   /* DEFINE RANDOM NUMBER GENERATOR GIVING VALUES
1  003C   00073            0.0 <= RANDOM <= 1000.0                              */
1  003C   00074  FUNCTION RANDOM(VAR X:INTEGER): REAL;
2  0000   00075  BEGIN
2  004E   00076     X:=128 * X MOD 8963;
2  0066   00077     RANDOM:=X/ 8963.0 * 1000.0
2  007E   00078  END;  /*   RANDOM   */
1  0044   00079  BEGIN /* TEST3 */
1  0044   00080     X:=3142;
1  004C   00081     FOR I:=1 TO N DO A(I):=RANDOM(X);
1  00B2   00082     FOR I:=1 TO N DO WRITELN(' ',A(I):10:3);
1  011C   00083     WRITELN('-':1,'LARGEST VALUE=',MAX(A,N):10:3);
1  0172   00084  END;  /* TEST3 */
```

```
TEST3
   870.690
   448.399
   395.068
   568.783
   804.195
   936.963
   931.273
   202.945
   977.016
    58.128
   440.366
   366.841
   955.707
   330.470
   300.123
   415.709
   210.755
   976.682
    15.285
   956.488
   430.436
    95.838
   267.321
   217.115
   790.695

LARGEST VALUE=   977.016
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          PASCAL'S TRIANGLE

0  0034   00085  PROCEDURE PTRIANGLE;
1  0000   00086  CONST N=10;
1  003C   00087  VAR X:ARRAY(1..N,1..11) OF INTEGER;
1  005C   00088     I,J,K:INTEGER;
1  005C   00089  BEGIN
1  0074   00090     WRITELN('0':1,'PASCAL''S TRIANGLE':72);
1  0098   00091     FOR I:=1 TO N DO BEGIN  /* INITIALIZE TABLE */
1  00BC   00092        X(I,1):=1;  X(I,I+1):=0;
1  0126   00093     END;
1  012A   00094     FOR I:=2 TO N DO     /*  DETERMINE THE TRIANGLE  */
1  014E   00095        FOR J:=2 TO I DO BEGIN
1  0172   00096           K:=I-1;
1  017E   00097        X(I,J):=X(K,J-1)+X(K,J);
1  023A   00098        END;
1  0242   00099     /* WRITE THE TRIANGLE IN THE USUAL MANNER */
1  0242   00100     FOR I:=1 TO N DO BEGIN
1  0266   00101        WRITE('0':1);
1  0278   00102        FOR J:=1 TO I DO WRITE(X(I,J):10);
1  02EA   00103        WRITELN;
1  02F8   00104     END;
1  02FC   00105    /* WRITE THE TRIANGLE AS A TRIANGLE CENTERED ON THE PAGE */
1  02FC   00106     K:=12;
1  0304   00107     FOR I:=1 TO N DO BEGIN
1  0328   00108        J:=K-I;  WRITE('0':1);
1  0346   00109        WHILE J <> 0 DO BEGIN
1  0352   00110           WRITE(' ':5);  J:=PRED(J);
1  036E   00111        END;
1  0372   00112        FOR J:=1 TO I DO WRITE(X(I,J):10);
1  03E4   00113        WRITELN;
1  03F2   00114     END;
1  03F6   00115  END; /* PASCAL'S TRIANGLE */
```

PTRIANGLE

PASCAL'S TRIANGLE

```
                                1
                              1   1
                            1   2   1
                          1   3   3   1
                        1   4   6   4   1
                      1   5  10  10   5   1
                    1   6  15  20  15   6   1
                  1   7  21  35  35  21   7   1
                1   8  28  56  70  56  28   8   1
              1   9  36  84 126 126  84  36   9   1
                                    1
                                  1   1
                                1   2   1
                              1   3   3   1
                            1   4   6   4   1
                          1   5  10  10   5   1
                        1   6  15  20  15   6   1
                      1   7  21  35  35  21   7   1
                    1   8  28  56  70  56  28   8   1
                  1   9  36  84 126 126  84  36   9   1
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          SIMULATE WAR GAME ***RISK***

0  0034   00116   /* RISK WAR GAME - ATTACKER THROWS 3 DICE AND DEFENDER THROWS 2     */
0  0034   00117  PROCEDURE RISKGAME;
1  0000   00118  CONST N=1000;  START=5161;
1  003C   00119  VAR TABLE:ARRAY(1..5,1..6) OF INTEGER;
1  005C   00120     TOT:ARRAY(1..6) OF INTEGER;
1  006C   00121     B:ARRAY(1..5) OF INTEGER;
1  007C   00122     I,J,K,L,M,X,A,D:INTEGER;
1  007C   00123     TEST1,PP:REAL;
1  007C   00124  FUNCTION DICEROLL(VAR X:INTEGER):INTEGER;   /* DICE VALUE FROM 1 TO 6 */
2  0000   00125  BEGIN
2  004E   00126     X:=128 * X MOD 8963;
2  0066   00127     DICEROLL:=6 * X DIV 8963 + 1;
2  0082   00128  END;
1  00B4   00129  BEGIN
1  00B4   00130     X:=START;  A:=0;  D:=0;
1  00C8   00131     FOR J:=1 TO 6 DO BEGIN
1  00EC   00132        TOT(J):=0;
1  0110   00133        FOR I:=1 TO 5 DO TABLE(I,J):=0;
1  0176   00134     END;
1  017A   00135     /* SIMULATE N CONFRONTATIONS BETWEEN COMBATANTS */
1  017A   00136     FOR I:=1 TO N DO BEGIN
1  019E   00137        FOR J:=1 TO 5 DO BEGIN   /* GENERATE 5 DICE VALUES */
1  01C2   00138           K:=DICEROLL(X);  B(J):=K;
1  020A   00139           TABLE(J,K):=TABLE(J,K)+1;
1  0286   00140        END;
1  028A   00141        FOR J:=1 TO 2 DO BEGIN  /* USE BUBBLE SORT ON ATTACKER'S 3 DICE */
1  02AE   00142           L:=3 -J;
1  02BA   00143           FOR K:=1 TO L DO
1  02DE   00144              IF B(K) < B(K+1) THEN BEGIN
1  032E   00145                 M:=B(K);  B(K):=B(K+1);  B(K+1):=M;
1  03CA   00146              END;
1  03CA   00147        END;
1  03D2   00148        IF B(4) < B(5) THEN BEGIN  /* SORT DEFENDERS DICE */
1  03F0   00149           M:=B(4);  B(4):=B(5);  B(5):=M;
1  0428   00150        END;
1  0428   00151        FOR J:=1 TO 2 DO   /* FIGHT - DEFENDER WINS DRAWS */
1  044C   00152           IF B(J) > B(J+3) THEN A:=A+1 ELSE D:=D+1;
1  04BC   00153     END;
1  04C0   00154     WRITELN('0':1,'RISK':65);  WRITELN('0':1);
1  04F6   00155     WRITELN('-':1,'DICE VALUE':68);
1  051A   00156     WRITELN('0':1,'DICE #':45,'1':8,'2':8,'3':8,'4':8,'5':8,'6':8);
1  05AA   00157     FOR J:=1 TO 5 DO BEGIN
1  05CE   00158        WRITE('0':1,' ':41,J:1,' ':5);
1  0616   00159        FOR K:=1 TO 6 DO WRITE(TABLE(J,K):8);
1  0688   00160        WRITELN;
1  0696   00161     END;
1  069A   00162     FOR J:=1 TO 6 DO
1  06BE   00163        FOR K:=1 TO 5 DO
1  06E2   00164           TOT(J):=TOT(J) + TABLE(K,J);
1  076C   00165     WRITELN('0':1,'TOTALS':43,' ':4,TOT(1):8,TOT(2):8,TOT(3):8,
1  07EC   00166             TOT(4):8,TOT(5):8,TOT(6):8);
1  083A   00167     WRITELN('-':1,' ':40,'AFTER 1000 CONFRONTATIONS:');
1  085E   00168     WRITELN('-':1,' ':50,'THE ATTACKER LOST',D:8,' MEN');
1  08CA   00169     WRITELN('-':1,' ':50,'THE DEFENDER LOST',A:8,' MEN');
1  0924   00170     WRITE('-':1,' ':20,'THEREFORE IT IS BETTER TO');
1  095A   00171     IF A < D THEN WRITELN('DEFEND THAN ATTACK')
1  0978   00172     ELSE WRITELN('ATTACK THAN DEFEND');
1  098E   00173  END; /* RISK GAME */
```

ISKGAME

RISK

DICE VALUE

DICE #	1	2	3	4	5	6
1	165	151	165	182	191	145
2	162	168	163	174	168	165
3	188	134	157	166	187	168
4	167	167	164	166	172	164
5	167	175	175	152	164	167
TOTALS	849	795	824	840	882	810

AFTER 1000 CONFRONTATIONS:

THE ATTACKER LOST 919 MEN

THE DEFENDER LOST 1081 MEN

THEREFORE IT IS BETTER TO ATTACK THAN DEFEND

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          DETERMINE LETTER FREQUENCIES IN SAMPLE

0  0034   00174  PROCEDURE TCHAR;
1  0000   00175     VAR N,I,J:INTEGER;
1  003C   00176        TESTCHR:CHAR;
1  003C   00177        NBRCHR:ARRAY(1..255) OF SHRTINT;
1  004C   00178  BEGIN
1  005C   00179     FOR I:=1 TO 255 DO NBRCHR(I):=0;
1  00A8   00180     WRITELN('-':1,'SAMPLE TEXT TO BE ANALYSED':52);  WRITELN('-':1);
1  00A8   00181     READLN(N);
1  00F0   00182     FOR I:=1 TO N DO BEGIN
1  0114   00183        WRITE(' ':26);
1  0126   00184        FOR J:=1 TO 80 DO BEGIN
1  014A   00185           READ(TESTCHR:1);  WRITE(TESTCHR:1);
1  016E   00186           NBRCHR(ORD(TESTCHR)):=NBRCHR(ORD(TESTCHR))+1;
1  01BE   00187        END;
1  01C2   00188        WRITELN;
1  01D0   00189     END;
1  01D4   00190     WRITELN('0':1,'TABLE OF LETTER FREQUENCIES':30,
1  01F8   00191        'LETTER':15,'OCCURRENCES':20);
1  021C   00192     FOR I:=ORD('A') TO ORD('I') DO WRITELN(CHR(I):44,NBRCHR(I):20);
1  0222   00193     FOR I:=ORD('J') TO ORD('R') DO WRITELN(CHR(I):44,NBRCHR(I):20);
1  0300   00194     FOR I:=ORD('S') TO ORD('Z') DO WRITELN(CHR(I):44,NBRCHR(I):20);
1  0372   00195  END; /* TCHAR */
```

TCHAR

SAMPLE TEXT TO BE ANALYSED

AND EVERY TIME THEY FOUND OUT WHAT SEEMED TO BE A PURPOSE OF THEMSELVES, THE PURPOSE SEEMED SO LOW THAT THE CREATURES WERE FILLED WITH DISGUST AND SHAME. AND, RATHER THAN SERVE SUCH A LOW PURPOSE, THE CREATURES WOULD MAKE A MACHINE TO SERVE IT. THIS LEFT THE CREATURES FREE TO SERVE HIGHER PURPOSES.

TABLE OF LETTER FREQUENCIES

LETTER	OCCURRENCES
A	16
B	1
C	5
D	9
E	45
F	5
G	2
H	17
I	8
J	0
K	1
L	7
M	7
N	6
O	14
P	8
Q	0
R	19
S	21
T	24
U	12
V	5
W	6
X	0
Y	2
Z	0

MANITOBA PASCAL VERSION 1 (JUNE 1975) SIMPSON'S METHOD FOR FINDING AREA

```
0  0034   00196   /*  OPTIMIZED SIMPSON'S   */
0  0034   00197  FUNCTION FX(X:LONGREAL):LONGREAL; FORWARD;
0  0034   00198  FUNCTION SIMP(A,B,EPS:LONGREAL;  FUNCTION FX:LONGREAL): LONGREAL;
1  0000   00199     VAR T1,T2,A1,A2,H:LONG REAL;
1  0072   00200         I,N:INTEGER;
1  0072   00201  BEGIN
1  0072   00202     N:=2;  H:=(B-A)/2.0;
1  0082   00203     T1:=FX(A) + FX(B);  T2:=4.0*FX(A+H);
1  010A   00204     A2:=H/3.0*(T1+T2);
1  0126   00205     REPEAT
1  0126   00206        A1:=A2;  N:=2*N;  H:=(B-A)/N;
1  0160   00207        T1:=T1 + T2/2.0;  T2:=0.0;  I:=1;
1  018A   00208        WHILE I <= N-1 DO BEGIN
1  019A   00209           /* SUM POINTS INTRODUCED THIS ITERATION */
1  019A   00210           T2:=T2 + FX(A+I*H);  I:=I+2;
1  01EA   00211        END;
1  01EE   00212        T2:=4.0*T2;  A2:=H/3.0*(T1+T2)
1  0214   00213     UNTIL ABS(A2-A1) < EPS;
1  022C   00214     SIMP:=A2
1  022C   00215  END;  /*  SIMP  */
0  0034   00216
0  0034   00217   /*-----------------------------------------------------------------*/
0  0034   00218  FUNCTION SIMPD1(X:LONGREAL):LONGREAL;
1  0000   00219  BEGIN
1  004E   00220     SIMPD1:=X*LN(1.0+X)
1  005A   00221  END;  /* SIMPD1 */
0  0034   00222
0  0034   00223   /*-----------------------------------------------------------------*/
0  0034   00224  FUNCTION SIMPD2(X:LONGREAL):LONGREAL;
1  0000   00225  BEGIN
1  004E   00226     SIMPD2:=SIN(X/2.0)
1  005C   00227  END;  /* SIMPD2 */
0  0034   00228
0  0034   00229   /*-----------------------------------------------------------------*/
0  0034   00230  PROCEDURE TESTSIMP;
1  0000   00231  VAR A,B,EPS:LONGREAL;
1  003C   00232  BEGIN
1  003C   00233     A:=0.0;  B:=1.0;  EPS:=1.0E-7;
1  0060   00234     WRITELN('-':1,'AREA UNDER X*LN(1+X) BETWEEN',A:10:5,'AND',B:10:5,'=',
1  00CC   00235           SIMP(A,B,EPS,SIMPD1):10:5);
1  010E   00236     WRITELN('-':1,'AREA UNDER SIN(X/2) BETWEEN',A:10:5,'AND',B:10:5,'=',
1  017A   00237           SIMP(A,B,EPS,SIMPD2):10:5);
1  01BC   00238  END;  /* TESTSIMP */
```

```
TESTSIMP

AREA UNDER X*LN(1+X) BETWEEN   0.00000  AND   1.00000  =   0.25000

AREA UNDER SIN(X/2) BETWEEN   0.00000  AND   1.00000  =   0.24483
```

```
0  0034   00239  PROCEDURE TPERSON1;
1  0000   00240     VAR P,Q:PERSON;
1  003C   00241     MSP,MSQ:STATUS;
1  003C   00242     CS:ARRAY(STATUS) OF STRING(8);
1  003C   00243  BEGIN
1  005C   00244     CS(MARRIED):='MARRIED';  CS(WIDOWED):='WIDOWED';
1  007C   00245     CS(DIVORCED):='DIVORCED';  CS(SINGLE):='SINGLE';
1  0092   00246   /* LOAD ALL FIELDS OF RECORD FOR PERSON P */
1  00A2   00247     P.N.LAST:='WOODYARD';
1  00BA   00248     P.N.FIRST:='EDWARD';
1  00CE   00249     P.SINNBR:=612043795;
1  00DE   00250     P.SEX:=MALE;
1  00EE   00251     P.BIRTH.MO:=AUG;
1  00FE   00252     P.BIRTH.DAY:=30;
1  0112   00253     P.BIRTH.YEAR:=1941;
1  0126   00254     P.DEPENDENTS:=1;
1  0136   00255     P.MS:=SINGLE;
1  0146   00256     P.INDEPENDENTRES:=TRUE;
1  0156   00257   /* LOAD ALL FIELDS OF RECORD FOR PERSON Q */
1  0156   00258     WITH Q,N,BIRTH DO BEGIN
1  0172   00259        LAST:='BROWN';
1  018A   00260        FIRST:='SUSAN';
1  019E   00261        SINNBR:=214132654;
1  01AE   00262        SEX:=FEMALE;
1  01BE   00263        MO:=JUNE;
1  01CA   00264        DAY:=15;
1  01DA   00265        YEAR:=1943;
1  01EA   00266        DEPENDENTS:=1;
1  01FA   00267        MS:=DIVORCED;
1  020A   00268        DDATE.MO:=SEPT;
1  021A   00269        DDATE.DAY:=10;
1  022E   00270        DDATE.YEAR:=1971;
1  0242   00271        FIRSTD:=TRUE;
1  0252   00272     END;
1  0252   00273   /* CHECK MARRIAGE COMPATIBILITY */
1  0252   00274     FOR MSP:=MARRIED TO SINGLE DO BEGIN
1  0278   00275        P.MS:=MSP;
1  0288   00276        FOR MSQ:=SINGLE DOWNTO MARRIED DO
1  02AC   00277           WITH Q DO BEGIN
1  02B4   00278              MS:=MSQ;
1  02C4   00279              WRITE(' MARRIAGE BETWEEN    ',P.N.FIRST:9,P.N.LAST:9,
1  02FE   00280              '(',CS(P.MS),')    AND    ',N.FIRST:9,N.LAST:9,
1  0332   00281              '(',CS(MS),') IS ');
1  03B2   00282              IF (P.SEX¬=SEX)&((P.MS¬=MARRIED)&(MS¬=MARRIED))
1  03EC   00283                 THEN WRITELN('A POSSIBILITY')
1  03FE   00284                 ELSE WRITELN('FORBIDDEN');
1  0402   00285           END;
1  0418   00286     END;
1  041C   00287  END; /* TPERSON1 */
```

```
TPERSON1
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  MARRIED   )    AND    SUSAN    BROWN      (  SINGLE    ) IS   FORBIDDEN
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  MARRIED   )    AND    SUSAN    BROWN      (  DIVORCED  ) IS   FORBIDDEN
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  MARRIED   )    AND    SUSAN    BROWN      (  WIDOWED   ) IS   FORBIDDEN
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  MARRIED   )    AND    SUSAN    BROWN      (  MARRIED   ) IS   FORBIDDEN
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  WIDOWED   )    AND    SUSAN    BROWN      (  SINGLE    ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  WIDOWED   )    AND    SUSAN    BROWN      (  DIVORCED  ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  WIDOWED   )    AND    SUSAN    BROWN      (  WIDOWED   ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  WIDOWED   )    AND    SUSAN    BROWN      (  MARRIED   ) IS   FORBIDDEN
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  DIVORCED  )    AND    SUSAN    BROWN      (  SINGLE    ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  DIVORCED  )    AND    SUSAN    BROWN      (  DIVORCED  ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  DIVORCED  )    AND    SUSAN    BROWN      (  WIDOWED   ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  DIVORCED  )    AND    SUSAN    BROWN      (  MARRIED   ) IS   FORBIDDEN
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  SINGLE    )    AND    SUSAN    BROWN      (  SINGLE    ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  SINGLE    )    AND    SUSAN    BROWN      (  DIVORCED  ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  SINGLE    )    AND    SUSAN    BROWN      (  WIDOWED   ) IS   A POSSIBILITY
MARRIAGE BETWEEN    EDWARD    WOODYARD    (  SINGLE    )    AND    SUSAN    BROWN      (  MARRIED   ) IS   FORBIDDEN
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          DEMONSTRATION OF SET OPERATIONS

0  0034   00288  PROCEDURE TESTSETS;
1  0000   00289  TYPE COLOURSET=SET OF COLOUR;
1  0044   00290  VAR HUE1,HUE2,HUE3:COLOURSET;
1  0044   00291     NBR1,NBR2,NBR3:SET OF 0..31;
1  005C   00292     C1,C2:COLOUR;
1  005C   00293     C:ARRAY(COLOUR) OF STRING(10);
1  005C   00294     I:0..31;
1  006C   00295  PROCEDURE MEMOFSET(X:COLOURSET);
2  0000   00296  VAR M:COLOUR;
2  004E   00297  BEGIN
2  004E   00298     FOR M:=WHITE TO BLACK DO IF M IN X THEN WRITE(C(M):11); WRITELN;
2  00BA   00299  END; /* MEMOFSET */
1  007C   00300  BEGIN
1  007C   00301     C(WHITE):='WHITE';  C(YELLOW):='YELLOW';  C(ORANGE):='ORANGE';
1  0098   00302     C(PINK):='PINK';  C(RED):='RED';  C(BLUE):='BLUE';
1  0108   00303     C(PURPLE):='PURPLE';  C(BROWN):='BROWN';  C(BLACK):='BLACK';
1  0150   00304     C1:=BLACK;  HUE1:=(.RED..BROWN,WHITE.);
1  0172   00305     HUE2:=(.C1,YELLOW..BLUE.);
1  0192   00306     WRITE('SET HUE1 CONTAINS:');  MEMOFSET(HUE1);
1  01C0   00307     WRITE('SET HUE2 CONTAINS:');  MEMOFSET(HUE2);
1  01EE   00308     WRITE('UNION OF SETS:');  MEMOFSET(HUE1 + HUE2);
1  0228   00309     WRITE('INTERSECTION OF SETS:');  MEMOFSET(HUE1 * HUE2);
1  0262   00310     HUE3:=HUE1 - HUE2;
1  027E   00311     WRITE('SET DIFFERENCE(HUE1 - HUE2):');  MEMOFSET(HUE3);
1  02AC   00312     WRITE('SET HUE3 CONTAINS:');  MEMOFSET(HUE3);
1  02DA   00313     WRITELN('HUE1 IS EQUAL TO HUE2:',HUE1=HUE2);
1  0310   00314    /* TEST SET INCLUSION */
1  0310   00315     WRITELN('HUE1 <= HUE3:',HUE1 <= HUE3);
1  034A   00316     WRITELN('HUE3 >= HUE1:',HUE3 >= HUE1);
1  0384   00317     WRITELN('HUE1 >= HUE3:',HUE1 >= HUE3);
1  03BE   00318     WRITELN('HUE3 <= HUE1:',HUE3 <= HUE1);
1  03F8   00319     NBR1:=(.0,3..7,10.);
1  0426   00320     WRITE('SET CONTAINS:');
1  0438   00321     FOR I:=0 TO 31 DO
1  046C   00322        IF I IN NBR1 THEN WRITE(I:5);
1  0494   00323     WRITELN;
1  04A6   00324  END;  /* TESTSETS */
```

```
 TESTSETS
SET HUE1 CONTAINS: WHITE      RED        BLUE       PURPLE     BROWN
SET HUE2 CONTAINS: YELLOW     ORANGE     PINK       RED        BLUE       BLACK
UNION OF SETS: WHITE      YELLOW     ORANGE     PINK       RED        BLUE       PURPLE     BROWN      BLACK
INTERSECTION OF SETS: RED        BLUE
SET DIFFERENCE(HUE1 - HUE2): WHITE      PURPLE     BROWN
SET HUE3 CONTAINS: WHITE      PURPLE     BROWN
HUE1 IS EQUAL TO HUE2:  FALSE
HUE1 <= HUE3:  FALSE
HUE3 >= HUE1:  FALSE
HUE1 >= HUE3:   TRUE
HUE3 <= HUE1:   TRUE
SET CONTAINS:    0    3    4    5    6    7   10
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          FAMILY TREE TO TEST POINTERS

0  0034   00325  PROCEDURE TPOINT1;
1  0000   00326  TYPE SEX=(MALE,FEMALE);
1  003C   00327     PERSON=RECORD
1  003C   00328               NAME,FIRSTNAME:STRING(15);
1  003C   00329               AGE:INTEGER;
1  003C   00330               MARRIED:BOOLEAN;
1  003C   00331               FATHER,CHILD:@PERSON;
1  003C   00332               CASE S:SEX OF
1  003C   00333                  MALE:(ENLISTED,BOLD:BOOLEAN);
1  003C   00334                  FEMALE:(PREGNANT:BOOLEAN;
1  003C   00335                          SIZE:ARRAY(1..3) OF INTEGER)
1  004C   00336            END;
1  004C   00337  VAR X:ARRAY(-10..5,COLOUR,WEEK) OF @PERSON;
1  005C   00338  BEGIN
1  0084   00339     NEW(X(-4,BLUE,MONDAY));
1  00A4   00340     WITH X(-4,BLUE,MONDAY)@ DO BEGIN
1  00BC   00341        NAME:='BROWN';  FIRSTNAME:='ALEX';  AGE:=24;  NEW(FATHER);
1  0110   00342        WITH FATHER@ DO BEGIN
1  0120   00343           NAME:='BROWN';  FIRSTNAME:='ROBERT';  AGE:=56;  NEW(FATHER);
1  0174   00344           WITH FATHER@ DO BEGIN
1  0184   00345              NAME:='BROWN';  FIRSTNAME:='LIONEL';  AGE:=80;
1  01C0   00346           END;
1  01C0   00347        END;
1  01C0   00348     END;
1  01C0   00349   /* FAMILY TREE */
1  01C0   00350     WITH X(-4,BLUE,MONDAY)@ DO
1  01D8   00351        WRITELN(FIRSTNAME,NAME,'IS',AGE:5,'YEARS OLD');
1  023A   00352     WITH X(-4,BLUE,MONDAY)@.FATHER@ DO  /* FATHER */
1  025A   00353        WRITELN(FIRSTNAME,NAME,'IS',AGE:5,'YEARS OLD');
1  0282   00354     WITH X(-4,BLUE,MONDAY)@.FATHER@.FATHER@ DO /* GRANDFATHER */
1  02E4   00355        WRITELN(FIRSTNAME,NAME,'IS',AGE:5,'YEARS OLD');
1  0346   00356  END;  /* TPOINT1 */
```

```
 TPOINT?
ALEX            BROWN           IS   24  YEARS OLD
ROBERT          BROWN           IS   56  YEARS OLD
LIONEL          BROWN           IS   80  YEARS OLD
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          DEMO OF FOR, WHILE, REPEAT AND CASE STAT

0  0034   00357  PROCEDURE TEST4;
1  0000   00358     TYPE WKWEEK=MONDAY..FRIDAY;
1  004C   00359     VAR X:REAL;
1  004C   00360        M,N,O:WKWEEK;
1  004C   00361        WK:ARRAY(WKWEEK) OF STRING(10);
1  004C   00362        I:INTEGER;
1  004C   00363  FUNCTION F(VAR I:INTEGER):BOOLEAN;
2  0000   00364  BEGIN
2  004E   00365     I:=I+1;  WRITE(I,FALSE:2);  F:=FALSE
2  007E   00366  END;
1  005C   00367  FUNCTION T(VAR I:INTEGER):BOOLEAN;
2  0000   00368  BEGIN
2  004E   00369     I:=I+1;  WRITE(I,TRUE:2);  T:=TRUE
2  007E   00370  END;
1  005C   00371  BEGIN
1  005C   00372     WK(MONDAY):='MONDAY';  WK(TUESDAY):='TUESDAY';
1  0088   00373     WK(WEDNESDAY):='WEDNESDAY'; WK(THURSDAY):='THURSDAY';
1  00B2   00374     WK(FRIDAY):='FRIDAY';
1  00CA   00375   /* FOR STATEMENTS */
1  00CA   00376     WRITE(' ':1);
1  00CA   00377     FOR I:=-10 TO 5 DO WRITE(I:10);  WRITELN;
1  0124   00378     FOR I:=-10 DOWNTO -20 DO WRITE(I:10);  WRITELN;
1  016A   00379     FOR M:=MONDAY TO THURSDAY DO WRITE(WK(M):15);  WRITELN;
1  01C4   00380     N:=TUESDAY;
1  01CC   00381     FOR M:=FRIDAY DOWNTO N DO WRITE(WK(M):15);  WRITELN;
1  0224   00382     M:=THURSDAY;
1  022C   00383     FOR O:=M DOWNTO N DO WRITE(WK(O):15);  WRITELN;
1  0282   00384   /* WHILE STATEMENTS */
1  0282   00385     X:=-2.0;  WRITE(' ':1);
1  029C   00386     WHILE X <= 5.0 DO BEGIN
1  02A8   00387        WRITE(X);  X:=X+0.25;
1  02C6   00388     END;  WRITELN;
1  02D8   00389     M:=FRIDAY;  WRITE(' ':1);
1  02F2   00390     WHILE M>MONDAY DO BEGIN
1  02FE   00391        WRITE(WK(M):15);  M:=PRED(M);
1  0334   00392     END;  WRITELN;
1  0346   00393   /* REPEAT STATEMENTS */
1  0346   00394     X:=-2.0;  WRITE(' ':1);
1  034E   00395     REPEAT
1  0360   00396        WRITE(X);  X:=X+0.25;
1  037E   00397     UNTIL X>5.0;  WRITELN;
1  0398   00398     M:=FRIDAY;  WRITE(' ':1);
1  03B2   00399     REPEAT
1  03B2   00400        WRITE(WK(M):15);  M:=PRED(M);
1  03E8   00401     UNTIL M=MONDAY;  WRITELN;
1  0402   00402   /* CASE STATEMENTS */
1  0402   00403     X:=1.246;
1  040A   00404     FOR I:=0 TO 10 DO BEGIN
1  042E   00405        WRITE(I);
1  0440   00406        CASE I OF
1  0440   00407           3: WRITE('SIN OF',X,'=',SIN(X));
1  04B2   00408           7,2: WRITE('COS OF',X,'=',COS(X));
1  0506   00409           5,1: WRITE('EXP OF',X,'=',EXP(X));
1  0506   00410           9: WRITE('SQR OF',X,'=',SQR(X))
1  05A6   00411        END;
1  05C0   00412        WRITELN;
1  05CE   00413     END;
1  05D2   00414     FOR M:=FRIDAY DOWNTO MONDAY DO BEGIN
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          DEMO OF FOR, WHILE, REPEAT AND CASE STAT

1  05F6   00415         WRITE(WK(M):11);
1  0618   00416         CASE M OF
1  0618   00417            WEDNESDAY,FRIDAY: WRITE('FIRST':15,WK(M):11);
1  066E   00418            MONDAY,THURSDAY: WRITE('SECOND':15,WK(M):11)
1  06A2   00419         END;
1  06B4   00420         WRITELN;
1  06C2   00421      END;
1  06C6   00422    /* DEMONSTRATE LOGICAL OPERATORS */
1  06C6   00423      WRITELN('-':1);  I:=0;
1  06DE   00424      WRITELN(T(I) AND F(I) AND T(I));
1  0762   00425      I:=0;
1  0768   00426      WRITELN(¬(F(I)|T(I))|¬F(I));
1  07EC   00427      I:=0;
1  07F2   00428      WRITELN(¬(T(I)&T(I)&¬(¬T(I)|F(I))));
1  089A   00429      I:=0;
1  08A0   00430      WRITELN(¬(F(I)&T(I)&¬(¬T(I)|F(I))));
1  0948   00431   END;  /* TEST4 */
```

```
TEST4
     -10        -9        -8        -7        -6        -5        -4        -3        -2        -1         0         1         2
       3         4         5
     -10       -11       -12       -13       -14       -15       -16       -17       -18       -19       -20
   MONDAY          TUESDAY         WEDNESDAY       THURSDAY
   FRIDAY          THURSDAY        WEDNESDAY       TUESDAY
   THURSDAY        WEDNESDAY       TUESDAY
    -2.000000E 00      -1.750000E 00      -1.500000E 00      -1.250000E 00      -1.000000E 00      -7.500000E-01      -5.000000E-01
    -2.500000E-01       0.000000E-01       2.500000E-01       5.000000E-01       7.500000E-01       1.000000E 00       1.250000E 00
     1.500000E 00       1.750000E 00       2.000000E 00       2.250000E 00       2.500000E 00       2.750000E 00       3.000000E 00
     3.250000E 00       3.500000E 00       3.750000E 00       4.000000E 00       4.250000E 00       4.500000E 00       4.750000E 00
     5.000000E 00
    FRIDAY          THURSDAY        WEDNESDAY       TUESDAY
    -2.000000E 00      -1.750000E 00      -1.500000E 00      -1.250000E 00      -1.000000E 00      -7.500000E-01      -5.000000E-01
    -2.500000E-01       0.000000E-01       2.500000E-01       5.000000E-01       7.500000E-01       1.000000E 00       1.250000E 00
     1.500000E 00       1.750000E 00       2.000000E 00       2.250000E 00       2.500000E 00       2.750000E 00       3.000000E 00
     3.250000E 00       3.500000E 00       3.750000E 00       4.000000E 00       4.250000E 00       4.500000E 00       4.750000E 00
     5.000000E 00
    FRIDAY          THURSDAY        WEDNESDAY       TUESDAY
           0
           1  EXP OF       1.245999E 00  =       3.476407E 00
           2  COS OF       1.245999E 00  =       3.191164E-01
           3  SIN OF       1.245999E 00  =       9.477156E-01
           4
           5  EXP OF       1.245999E 00  =       3.476407E 00
           6
           7  COS OF       1.245999E 00  =       3.191164E-01
           8
           9  SQR OF       1.245999E 00  =       1.552514E 00
          10
IDAY                FIRST FRIDAY
URSDAY              SECOND THURSDAY
DNESDAY             FIRST WEDNESDAY
ESDAY
NDAY                SECOND MONDAY


           1 T              2 F  FALSE
           1 F              2 T              3 F   TRUE
           1 T              2 T              3 T              4 F  FALSE
           1 F   TRUE

EXECUTION TIME    1.0196  SECS
```

```
MANITOBA PASCAL  VERSION 1 (JUNE 1975)          MAIN PROGRAM CONSISTS OF PROCEDURE CALLS

0  0034   00432  BEGIN /*  MAIN PROGRAM */
0  0034   00433     WRITELN('1':1,'TEST1');  TEST1;  WRITELN('-':1,'TEST2');  TEST2;
0  0066   00434     WRITELN('1':1,'TEST3');  TEST3;
0  00CA   00435     WRITELN('1':1,'PTRIANGLE');  PTRIANGLE;
0  00FC   00436     WRITELN('1':1,'RISKGAME');  RISKGAME;
0  012E   00437     WRITELN('1':1,'TCHAR');  TCHAR;
0  0160   00438     WRITELN('1':1,'TESTSIMP');  TESTSIMP;
0  0192   00439     WRITELN('1':1,'TPERSON1');  TPERSON1;
0  01C4   00440     WRITELN('1':1,'TESTSETS');  TESTSETS;
0  01F6   00441     WRITELN('1':1,'TPOINT1');  TPOINT1;
0  0228   00442     WRITELN('1':1,'TEST4');  TEST4;
0  023A   00443  WRITELN('-':1,'EXECUTION TIME',CPUTIME:10:4,'SECS');
0  02A8   00444  END. /* MAIN PROGRAM */
COMPILE TIME:    2.454 SECOND(S)
PARSED SUCCESSFULLY
       NO WARNING(S) DETECTED
       NO ERROR(S) DETECTED
```

CHAPTER 4

USER MANUAL

Standard PASCAL is considered to be the language presented by Niklaus Wirth and Kathleen Jensen in "PASCAL User Manual and Report", (Jen74).

This chapter is not intended to be a complete description of the language PASCAL. It describes details relating to this particular implementation, and is an expansion of the topics discussed by Wirth and Jensen.

4.1 IDENTIFIERS

Identifiers must begin with a letter followed by any number of letters, digits and underscores (_). The underscore was allowed to make long identifiers more readable. The length of identifier is restricted only by the length of line because the end of line acts as a delimiter. Identifiers are significant for the first 32 characters, so that characters beyond this point do not contribute to identifier uniqueness.

Examples:

(Valid Identifiers)	(Invalid Identifiers)
X	$XY
Z79P2317W	P79,23
RATE_OF_PAY	7XY
P___X__1	_X

(Long Identifiers)

```
                                  32
1  THIS_IS_A_VERY_VERY_LONG_NAME  |
2  THIS_IS_A_VERY_VERY_LONG_NAME_ALSO
3  THIS_IS_A_VERY_VERY_LONG_NAME_ALSO_IS_IT_NOT
```

1 and 2 are different.

1 and 3 are different.

2 and 3 are the same identifier.

A procedure or function name must be distinct from all other procedure and function names in the first 8 characters, or the wrong subprogram may be called at run time.

4.2 COMMENTS

Comments are delimited by the sequence /* on the left and by */ on the right. The comment itself may contain any character sequence (except of course */ which would terminate the comment), and may be split over several cards. Comments act as separators and therefore should not be placed in the middle of keywords, identifiers or constants. The /* starting a comment must not begin in column 1 as it would then be taken as an end-of-file indicator by the System.

Examples:

```
/* THIS IS A HEADING        */
/* THIS IS A HEADING
   SPLIT OVER MORE THAN 1 LINE */
IF A > B/*HERE*/THEN X := X * Y;
```

4.3 NUMBERS

The syntax for numbers is standard. Two different precisions are provided for integers (SHORT INTEGER and INTEGER) and for reals (REAL and LONG REAL).

A constant without a decimal point or exponent is considered to be a full-word integer (INTEGER). If this constant is immediately followed by the letter S, a half-word integer (SHORT INTEGER) is established.

A constant containing a decimal point and/or exponent is taken as a single precision floating-point number (REAL) unless it is followed by the letter L, in which case a double precision floating-point number (LONG REAL) is constructed.

Examples:

(SHORT INTEGER)	(INTEGER)	(REAL)	(LONG REAL)
-85S	-87653142	-0.5	-0.5L
0S	0	9.8E-31	9.8E-31L
32767S	97	10E10	10E10L

4.4 HEXADECIMAL CONSTANTS

The hexadecimal constants are indicated by the symbol # followed by a sequence of hexadecimal digits. The hexadecimal number is treated as an INTEGER unless it is followed by a qualifier. The valid qualifiers are:

S - SHORT INTEGER
R - REAL
L - LONG REAL
X - CHAR (single character)

Examples:

(SHORT INTEGER)	(INTEGER)	(REAL)	(LONG REAL)	(CHAR)
#00S	#0F	#41100000R	#0L	#C1X
#100S	#41B0D254		#CE00000080000000L	
#1FFFS	#7FFFFFFF			

4.5 STRINGS

Variables of type STRING are designated by STRING(n) where 1 <= n <= 256. Character strings are delimited by single quote marks and a quote within the string is indicated by two quotes. The maximum length allowed for any character string is 256 characters. They may be split over several cards if necessary.

Examples:

```
'THIS IS A STRING'
'''A'''
'THIS IS A VERY LONG STRING AND EVEN HAS A '
'AT THE END OF THE LINE AND START OF THE
NEXT INDICATING A QUOTE WITHIN THE STRING'
```

CHAR and STRING Variables

Variables of type CHAR occupy a single byte. Operations are usually performed on the contents using registers (IC and STC instructions). STRING values are never placed in registers (MVC and CLC instructions are used). CHAR and STRING variables are compatible and may be compared with, or assigned to, each other.

Assignment

If the string on the left is longer than the one on the right, the string is padded out to full size with blanks.

If the string on the left is too short to hold the string on the right, it is truncated on the right.

Example:

```
VAR A : CHAR;  R, S : STRING(10);  T : STRING(15);
BEGIN
   A := 'HORSE';
   R := 'CAR';
   S := 'THIS IS A TEST CASE';
   T := S
END.
```

Contents of variables:

A	H
R	CAR␣␣␣␣␣␣␣
S	THIS␣IS␣A␣
T	THIS␣IS␣A␣␣␣␣␣␣

Comparison

If the CHAR or STRING operands to be compared are not of the same length, they are compared to the length of the shorter but a compile-time warning is given.

Examples:

Comparison		Result
'THIS IS A TEST'	'THIS'	=
'ABC'	'AB'	=
'ABC'	'ABA'	>

If a user desires a comparison to be to the length of the longer, then the shorter can be moved into a dummy variable of the correct length (causing padding on the right with blanks) before the comparison is made.

4.6 SCALARS

Scalar variables use a full-word of memory. The ordinal number of the first constant listed is 0, while the second is 1 and the N^{th} is N-1.

Complete compile-time checking is performed to ensure that all uses of scalar variables and scalar constants are valid.

Example: Assume the declarations:

```
TYPE COLOUR = (RED, GREEN, BLUE, BROWN);
     WEEK = (SUN, MON, TUES, WED, THURS, FRI, SAT);
     WORK_WEEK = MON..FRI;
VAR  A : COLOUR;
     B : WEEK;
     C : WORK_WEEK;
```

The assignments

```
A := RED;  B := MON;  B := C;
```

are valid because the assigned values are of compatible types and lie within the required ranges.

The assignments

```
A := MON;  A := B;  C := A;
```

are invalid because the values being assigned are not of the same scalar type.

4.7 SUBRANGES

The associated scalar type of a subrange may be any user-defined scalar as well as SHORT INTEGER, INTEGER, REAL, LONG REAL or BOOLEAN. A subrange of type CHAR or STRING is not allowed. No special action is taken for subranges of real values. The limits merely serve as comments.

The use of a subrange does not imply any saving in memory, but subranges of user-defined scalars imply exhaustive compile-time compatibility checks.

Compile-time checks for subranges of integers are made only in cases involving a constant. The limits of the subrange are used to check ranges at run time if the checking option is specified.

Example: Assume the declarations:

```
TYPE T_WEEK = (SUN, MON, TUES, WED, THURS, FRI, SAT);
     T_SUN_THURS = SUN..THURS;  T_TUES_SAT = TUES..SAT;
     T_MON_WED = MON..WED;      T_THURS_SAT = THURS..SAT;
VAR  A1_10 : 1..10;    B0_30 : 0..30;  C20_30 : 20..30;
     D_WEEK : T_WEEK;  D_SUN_THURS : T_SUN_THURS;
     D_TUES_SAT : T_TUES_SAT;  D_MON_WED : T_MON_WED;
     D_THURS_SAT : T_THURS_SAT;
```

The following assignments are allowed:

```
A1_10 := 3;  A1_10 := B0_30;  C20_30 := B0_30;

B0_30 := C20_30;  A1_10 := -100 + A1_10;

D_WEEK := WED;  D_SUN_THURS := D_MON_WED;

D_WEEK := D_SUN_THURS;
```

The assignments

```
D_MON_WED := D_THURS_SAT;  D_TUES_SAT := SUN;
```

are flagged as compile-time errors.

The assignments

```
A1_10 := -3;  B0_30 := 31;  C20_30 := 19;
```

result in compile-time error messages if the checking option is specified.

If run-time checking code is produced

```
A1_10 := -100 + A1_10;
```

results in a run-time error while the assignments

```
A1_10 := B0_30;  C20_30 := B0_30;
```

produce errors only if the value being assigned is outside the allowed range.

The assignments

```
D_SUN_THURS := D_WEEK;  D_TUES_SAT := D_MON_WED;
D_THURS_SAT := D_TUES_SAT;
```

are given warnings at compile time indicating that it is possible for an out-of-range value to be assigned at run time. Run-time checking code is produced if the checking option is in effect. This is the only case where the compiler allows the potentially dangerous use of a user-defined scalar or subrange of a user-defined scalar. This exception was made for the following reason. If the declaration

```
X : ARRAY (T_SUN_THURS, T_SUN_THURS) OF INTEGER;
```

is added to the example above, then

```
X(D_SUN_THURS, D_MON_WED) := 2;
```

is allowed while

```
X(D_WEEK, D_TUES_SAT) := 2;
```

results in a compile-time error for each subscript because it is possible for the subscript to have a value which is out of range at run time. If the variable D_WEEK contains a value which the user wishes to use as a subscript of X, it is almost impossible (without equivalencing variables of type T_WEEK and T_SUN_THURS using variant parts of records) to do this without this one exception mentioned above. It is for this reason that the assignments:

```
D_SUN_THURS := D_WEEK;  D_TUES_SAT := D_MON_WED;
D_THURS_SAT := D_TUES_SAT;
```

are allowed to proceed with only a warning being given. This type of assignment is checked at run time if the checking option is specified.

In summary, the language provides information for compile-time checking of assignments involving user-defined scalars and subranges of scalars. This information is used to eliminate run-time checks where these types are used. Run-time checks on PRED, SUCC and the assignments described in the last paragraph constitute the only cases that cannot be checked at compile time.

4.8 SETS

A set is allocated a full-word of memory. Each member of the set is represented by one bit position, restricting the number of elements in a set to 32.

The base type of a set must be one of the following:

a) User-defined scalar containing at most 32 constants.
b) Subrange (M..N) of a user-defined scalar where ORD(N) <= 31.
c) Subrange (M..N) of integers where M >= 0 and N <= 31.

Compile-time checking is performed for set operations. This checking consists of a comparison of the base types of the sets to see if they are compatible. It is possible for some set members to escape detection by the compiler because they are of the correct base type, even though they may be out of the allowed range. A check for invalid set members is made on assignment at run time if the checking option is specified.

Example: Assume the declarations:

```
TYPE TOY = (BALL, BAT, CAR, PLANE);
     WEEK = (SUN, MON, TUES, WED, THURS, FRI, SAT);
     WKWEEK = MON..FRI;
VAR  A_TOY : TOY;  B_WEEK : WEEK;  C_WKWEEK : WKWEEK;
     J1_10 : 1..10;  K : INTEGER;
     SET_WKWEEK : SET OF WKWEEK;
     SET1_10 : SET OF 1..10;
```

The symbols (. and .) enclose sets. The statements

```
SET_WKWEEK := (..);

SET_WKWEEK := SET_WKWEEK + (.C_WKWEEK, TUES.);

SET1_10 := (.J1_10, 3.);
```

compile correctly and do not cause run-time errors.

The statements

```
SET_WKWEEK := (.MON, SAT, B_WEEK.);

SET1_10 := (.J1_10, K, 85.);
```

compile correctly but will both cause errors at run time.

The statement

```
SET_WKWEEK := SET1_10;
```

is invalid because the base types of the sets are not compatible.

If a set is constructed by specifying the set members, the type of the first member is considered to be the base type of the set and the other members are checked for compatibility with it.

The statement

```
SET_WKWEEK := (.TUES, B_WEEK, C_WKWEEK, A_TOY, CAR, J1_10, K, 10);
```

causes error messages for members A_TOY, CAR, J1_10, K and 10 because they are incompatible with the scalar of which TUES is a constant.

If sets containing a single member as in

```
SET_WKWEEK := (.WED.);  SET_WKWEEK := SET_WKWEEK + (.C_WKWEEK.);
```

are indicated without the set brackets, as in

```
SET_WKWEEK := WED;  SET_WKWEEK := SET_WKWEEK + C_WKWEEK;
```

the compiler constructs the required sets and issues the warning SINGLETON SET.

4.9 POINTERS

Pointers occupy a full-word of memory. Only the simple forms of procedures NEW and DISPOSE are allowed. The tagfield values for a record with variant parts may not be specified. In this case, the space allocated is large enough to accomodate the largest alternative. Procedure DISPOSE is simply ignored, as no garbage collection is done.

The symbol @ is used to indicate a pointer rather than the standard ↑.

4.10 ARRAYS

Arrays are specified by giving the index type and component type. The allowed index types are user-defined scalars, subranges of integers or user-defined scalars, or BOOLEAN. The component type may be any definable type. The number of dimensions is not limited.

The following:

```
CONST N = 10;
TYPE VECTOR = ARRAY(1..N) OF INTEGER;
     MATRIX = ARRAY(1..N) OF VECTOR;
VAR  X : MATRIX;
     Y : ARRAY(1..N,1..N) OF INTEGER;
     Z : ARRAY(1..N) OF ARRAY(1..N) OF INTEGER;
```

shows the declaration of three N by N arrays X, Y and Z. Although these declarations are equivalent, the compiler does not consider them to be of exactly the same type. The assignments

```
X := Y;  Y(3) := Z(5);
```

are not allowed because the component types do not have the same type descriptors.

If the declarations are of the form:

```
CONST N = 10;
TYPE MATRIX = ARRAY(1..N,1..N) OF INTEGER;
VAR X,Y : MATRIX;  Z : MATRIX;
```

or of the form

```
VAR X,Y,Z : ARRAY(1..N,1..N) OF INTEGER;
```

then the type descriptors for X, Y and Z are identical and the assignments are allowed. In the case

```
X := Y;
```

the entire 100 elements of Y are stored in the corresponding

locations of X while

```
Y(3) := Z(5);
```

causes the 5^{th} row of Z to be assigned to the 3^{rd} row of Y.

Example:

```
CONST N = 10;
TYPE VECTOR = ARRAY(1..N) OF INTEGER;
VAR  X : ARRAY (1..20) OF VECTOR;
- - -
   FUNCTION SUM (VAR Y : VECTOR) : INTEGER;
   VAR I, VSUM : INTEGER;
   BEGIN
      VSUM := 0;
      FOR I := 1 TO N DO VSUM := VSUM + Y(I);
      SUM := VSUM
   END;
BEGIN
   - - -
   WRITELN(SUM(X(3)))
END.
```

This program passes X(3) whose component type is VECTOR to SUM for summation of the elements in this row of X. Similar to assignment, this technique can only be used when the formal and actual parameters have the same type descriptors.

A vector of CHAR is treated in the same manner as a character string for assignments, I/O and comparisons.

Example: Assume the declarations:

```
TYPE BUFFER = ARRAY(1..80) OF CHAR;
VAR  LINE : BUFFER;  I : INTEGER;
```

LINE can be read one character at a time by

```
FOR I := 1 TO 80 DO READ(LINE(I));
```

or it can be read as one character string by

```
READ(LINE);
```

which makes only one request to the input routine rather 80 requests.

LINE can be tested to see if it contains $JOB starting in column 1 simply by saying

```
IF LINE = '$JOB' THEN
```

which compares a string of length 80 with a string of length 4 and compares to the length of the shorter. A substring function is not available so this technique cannot be used to test a sequence of characters in the middle of a string.

4.11 RECORDS

Variant parts may be specified with or without tagfields. Each alternative mapping in a variant starts on a double-word boundary so that the effect of equivalencing variables of different types can be achieved.

Similar to arrays, record assignment is allowed if the types are identical.

Example: Assume the declarations:

```
TYPE OPT = 1..3;
     WEEK = (SUN, MON, TUES, WED, THURS, FRI, SAT);
     EQUIV = RECORD
                 CASE OPT OF
                    1: (W1 : WEEK;  I : INTEGER);
                    2: (W2 : MON..FRI;  J : 1..10);
                    3: (K  : INTEGER;  S : STRING(4))
             END;
VAR  A, B : EQUIV;
```

It is possible to circumvent compile-time and run-time checks and cause assignment of values which are out of range or of an invalid type by using variants.

The statements

```
A.W1 := SAT;  A.I := #C3C1E3E2;
WRITELN(A.K,A.S);
```

cause

6 CATS

to be printed. If, however, A.W2 or A.J are referred to, they contain values which are out of range, although this goes undetected. This use of variants is not recommended and should be used with extreme caution.

4.12 STATEMENTS

4.12.1 IF Statement

The two forms of the IF statement

IF expression THEN statement

and

IF expression THEN statement ELSE statement

are allowed. According to the language description, a semi-colon is never valid before the ELSE. If the compiler detects a semi-colon before an ELSE, it issues a warning and continues as if it had not been present.

4.12.2 WHILE Statement

The WHILE statement has the form:

WHILE expression DO statement

The expression is evaluated prior to each iteration. The statement is

repeatedly executed until the expression becomes FALSE. If the expression is FALSE on entry to the WHILE, the statement is not executed at all.

4.12.3 REPEAT Statement

The REPEAT statement has the form:

REPEAT statement { ; statement } UNTIL expression

The statements between REPEAT and UNTIL are executed repeatedly until the expression becomes TRUE. The statements are always executed at least once.

4.12.4 Assignment Statement

The assignment restrictions are indicated at the appropriate places in the preceding discussion of the various types.

In general, assignment is allowed if the types are compatible. Assignment of structured types is allowed only if both have the same type descriptor.

4.12.5 GOTO Statement

Branching to labels outside of the current procedure or function is not permitted. All labels used in the program must be declared. Labels longer than 4 digits are given warnings. Branches into structured statements are detected by the compiler, and are not allowed.

4.12.6 FOR Statement

The FOR statement has two forms:

1) FOR control variable := initial value TO final value DO S

2) FOR control variable := initial value DOWNTO final value DO S

The control variable must be predefined (i.e., it is not local to the FOR statement).

The initial and final value are evaluated only once on entry to the FOR statement. If the initial value exceeds the final value in form 1 or is less than the final value in form 2, the statement in the range of the FOR is not executed. In this case the control variable retains its previous value.

Upon normal exit from a FOR statement (i.e., the loop has been performed the required number of times), the control variable, rather than being undefined, remains at the last valid value assigned to it (i.e., the final value).

If an exit is taken from a FOR statement via a jump, the control variable remains at the most recent value assigned to it in the loop. If the type of the control variable is a user-defined scalar or subrange thereof, complete compile-time checking for compatibility with the initial and final values is performed. An error message is given if it is possible for an invalid value to be assigned.

If the control variable is a subrange of integers and the checking option is specified, run-time range checks are performed on entry to the FOR statement by comparing the initial and final values against the range limits of the control variable.

If a statement in the range of the FOR statement alters the initial or final value, it has no effect on the number of iterations in the loop and is not considered to be an error. If an assignment is made to the control variable, or if the control variable is an actual parameter in a call by reference within the range of the FOR, it is an error.

It is possible for a procedure or function, called within the range of the FOR, to modify the control variable by making an assignment to a global variable. This is not detected and should be avoided.

4.12.7 CASE Statement

A CASE selector may be of type INTEGER, BOOLEAN, user-defined scalar, or a subrange of any of these.

It is not necessary for the CASE labels to be specified in any particular order or for all the alternatives to be included.

If the CASE selector has a negative value, a run-time error message is given. If the CASE selector value is higher than the highest label used or is a value not covered by a label, processing continues with the next statement. If the CASE selector value matches a CASE label, a branch is taken to that alternative. Once this has been executed, processing continues with the statement following the CASE statement.

Care should be taken to avoid CASE statements with widely different labels.

For example:

```
CASE I OF
   1 : X := SIN(X);
 100 : X := COS(X);
1000 : X := EXP(X)
END;
```

would cause branch addresses to be established for all possible CASE index values between 0 and 1000. This would require approximately 2000 bytes of the program segment to be used for this purpose.

4.12.8 WITH Statement

The WITH statement serves to open the scope to a record variable, enabling the record fields to be accessed directly, without preceding them with the name of the record variable.

Example: Assume the declarations

```
VAR A : INTEGER;
    B : RECORD
           A : REAL;
           B : BOOLEAN
        END;
```

The variable A is known directly while the record fields are accessed by qualifying the record variable as in B.A and B.B . Inside the statement

```
WITH B DO S
```

the REAL variable A and BOOLEAN variable B can be accessed directly as in

```
A := 10.5;  B := TRUE;
```

but the record variable B is no longer known. The record fields can therefore not be specified as

```
B.A := 10.5;  B.B := TRUE;
```

inside the WITH statement.

4.13 FORWARD Declaration

Procedures and functions may be used before they are declared if they are "preannounced" by a FORWARD declaration. This facility allows mutually recursive procedures.

The FORWARD declaration also serves two other purposes. A check is not made to see that a procedure or function name, announced by a FORWARD declaration, is later included. Therefore, a FORWARD declaration can be used to make the name of a segment, which is to be included in the link-edit step, known to the compiler. This serves a similar purpose to the EXTERNAL declaration in other languages. Not only is the name available for linkage, but parameter number and type checking is performed on any calls of the external segment.

Parameter number and type checking cannot normally be performed when a procedure or function which is passed into a routine as a parameter is later called. The FORWARD declaration can be used to enable parameter checking to be performed in this case, if desired. When a formal parameter procedure or function is called, the compiler checks to see if the formal parameter name has been announced with a FORWARD declaration. If it has, and is of the correct type (procedure or function) and has the same result type (if a function), then parameter checking is performed using the information given in the FORWARD declaration.

Example:

In the program segment

```
PROCEDURE TEST (VAR X : INTEGER;  FUNCTION FX : REAL);
VAR Y : REAL;
BEGIN
   - - -
   Y := FX(Y,X);
```

parameter checking is not performed in the call of FX and this is indicated with a warning. If the above code sequence is preceded by

```
FUNCTION FX (A,B : REAL) : REAL; FORWARD;
```

then parameter number and type checking is performed, and in this example would detect that the actual parameter X is invalid because a REAL argument is required.

If a formal parameter procedure or function is called and the name is not announced by a FORWARD declaration, space is allocated for a maximum of 10 parameters.

4.14 FUNCTION Result Types

The result returned by a function may be any of SHORT INTEGER, INTEGER, REAL, LONG REAL, BOOLEAN, CHAR, SET, scalar, subrange or pointer. It may not be of type STRING, ARRAY or RECORD.

4.15 INPUT / OUTPUT

Only the standard input file SYSIN and output file SYSPRINT are supported. For this reason READ, READLN, EOLN and EOF always refer to SYSIN while WRITE and WRITELN always refer to SYSPRINT. The file names are not included when using these procedures and functions.

The I/O package implemented is not standard PASCAL. The changes and extensions were made to increase flexibility and to make I/O easier to use. For this reason programs written using standard I/O might require modification to have the desired effects when run using this compiler.

I/O is supported for seven different scalar types. These are:

SHORT INTEGER (SHRTINT)

INTEGER

REAL

LONG REAL

BOOLEAN (FALSE,TRUE)

CHAR - single character

STRING(n) (1<=n<=256) character string

4.15.1 OUTPUT

There are two output procedures:

WRITE(.....) adds the elements in the list to the output buffer and leaves the buffer pointer so that additional items can be moved into the buffer with a subsequent WRITE or WRITELN statement.

WRITELN(....) adds the elements in the list to the output buffer and then causes the buffer to be printed. Subsequent WRITE or WRITELN statements start to fill a new buffer. WRITELN without parameters causes the current output buffer to be printed.

The elements in the output lists are of one of the following forms:

e

e:f

e:f:d

where e is an expression and f and d, if specified, are unsigned integers.

If the form used is e it will be referred to as format-free I/O, while if the form is e:f or e:f:d it will be called formatted I/O.

The following printer control characters are recognized:

␣ (blank)	- single space
0	- double space
-	- triple space
+	- no space - overprint same line
1	- skip to new page

If a character other than these five is used as printer control, it is changed to a blank causing single spacing.

Format-Free Output

Each element in the output list is moved into the output buffer right-justified in a field of default width. The default width insures that the item printed is preceded by a minimum of 2 blanks. Thus the first

position of the buffer (printer control character) is always a blank causing single spacing.

The following table gives the default field widths used and also the number of elements of this type which will fit on one line of output.

Type	Default Width	# per Line
SHORT INTEGER	8	16
INTEGER	13	10
REAL	18	7
LONG REAL	26	5
BOOLEAN	7	18
CHAR	3	44
STRING(n)	n+2	

REAL and LONG REAL values are printed in floating-point form to maximum precision. BOOLEAN values are printed as TRUE or FALSE.

If there is insufficient room in the output buffer to accommodate an item in the output list, the buffer is printed and the item is moved in at the start of the new buffer. A long STRING (such as 256 characters) is printed at the beginning of a new line and is allowed to overflow into the next line.

Example: Format-Free Output

```
CONST PI=3.14159; N=10;
VAR X:LONG REAL; I:SHRTINT;
    A:CHAR; S:STRING(10); B:BOOLEAN;
BEGIN
 /* INTEGER VALUES */
  I:=-23S;
  WRITE(I,I+N);
  WRITELN( 16S,#7FFFFFFF,#10S,TRUNC(PI));
 /* REAL VALUES */
  X:=897.61354E23L;
  WRITE(X);
  WRITE(-7.2E-3,#41100000R);
  WRITELN;
 /* BOOLEANS, CHARACTERS AND STRINGS */
  A:='A'; S:='HORSE''S'; B:=TRUE;
  WRITELN(A,B,S,'SUM=',I>N,¬B,#E9X)
END.
```

This program produces 3 lines of output with single spacing:

1st Line

```
   -23          -13      16   2147483647      16          3
```

2nd Line

```
       8.9761354000000E 25     -7.199999E-03      1.000000E 00
```

3rd Line

```
 A   TRUE  HORSE'S     SUM=  FALSE  FALSE  Z
```

A blank was taken from the first position of each line for printer control.

Formatted Output

The form e:f can be used for any type of expression. The f indicates the field width to be used when printing the expression e. This field width is used in place of the default width which would be used for this type of expression in format-free output.

All elements are moved into the output buffer right-justified in the specified fields.

REAL values are printed to full precision in floating-point notation.

BOOLEAN values are printed as TRUE and FALSE if the field width is at least 5, otherwise they are printed as T and F.

The form e:f:d is only allowed for REAL expressions. The value is printed in a field of width f with d digits to the right of the decimal point in fixed-point notation. When this specification is used the last digit is rounded.

If the field width f is not large enough to hold an INTEGER or REAL value, the field is filled with asterisks.

If the field width specified for a STRING is less than the STRING size, the STRING is truncated on the right.

Example: Formatted Output

```
VAR I:INTEGER; K:SHRTINT;
    X:REAL; Z:LONG REAL;
    A:CHAR; S:STRING(10); B:BOOLEAN;
BEGIN
 /* INTEGER VALUES */
  I:=-23; K:=10S;
  WRITE(' ':1,I:5,K:3,697:5,ABS(I):3);
  WRITELN(#100:4,#FS:3,I:2);
 /* REAL VALUES */
  X:=-73.4966; Z:=987.3145967E-2L;
  WRITELN('0':1,X:14,X:10,X:10:3,Z:20, Z:8,Z:6:2);

 /* BOOLEANS,CHARACTERS AND STRINGS */
  A:='C'; S:='JOHNSON'; B:=FALSE;
  WRITELN('-':1,A:5,A:1,S:12,S:3,B:10,B:2)
END.
```

This program produces 3 lines of output. Printer control characters have been specified giving single, double and triple spacing respectively.

1st Line (single space)

```
 -23 10  697 23 256 15***
```

2nd Line (double space)

```
-7.349660E 01**********   -73.497 9.8731459670000E 00******** 9.87
```

3rd Line (triple space)

```
   CC  JOHNSON   JOH     FALSE F
```

4.15.2 INPUT

There are two input procedures:

READ(....) Reads from the input buffer and leaves the buffer pointer so that reading can continue from the same buffer with a subsequent READ or READLN statement.

READLN(....) Reads from the input buffer and then discards the remainder of the buffer. A subsequent READ or READLN would start reading from the next buffer. READLN without parameters causes the remainder of the current buffer to be discarded.

There are two functions available to aid in processing input:

EOLN - Has the value TRUE if the input buffer pointer is at the end of the current buffer, otherwise it has the value FALSE.

EOF - Has the value TRUE only if the file SYSIN is empty. Otherwise it has the value FALSE.

READLN always sets EOLN to FALSE and attempts to get the next input record so that EOF can be correctly set.

When execution begins EOLN is FALSE and EOF is TRUE or FALSE depending on whether the SYSIN file is empty or not.

Input-list specifications were designed to parallel those of output as closely as possible to provide a more flexible input capability than allowed in standard PASCAL.

Similar to output, elements of the input lists are of one of the following forms:

e

e:f

e:f:d

where e specifies the variable which is to be assigned the value read, and f and d are unsigned integers specifying format information.

Format-Free Input

The input list element is of the form e. The information can be placed in any columns of the input record.

The input routine scans the input record until it finds a non-blank character and then attempts to read a value of the required type.

INTEGER or REAL values must conform to the accepted specification. A blank, end-of-line or a character not valid in a number serves to delimit the number on the right. If a number of the correct type is not found, a run-time error message is given.

If the variable is of type BOOLEAN, the character T is read as TRUE while an F is taken as FALSE. Any other character is invalid and results in a run-time error message.

Character strings are delimited by quote marks ('). A quote within a string is represented by two quotes (''). If necessary, a string may be split over more than one record. If the input string is shorter than the declared string size, the string is padded on the right with blanks. If the input string is too long or the first non-blank character is not a quote, a run-time error message is given.

Example: Format-Free Input

```
VAR I:INTEGER; K:SHRTINT;
    X:REAL; Z:LONG REAL;
    A:CHAR; S:STRING(10); B:BOOLEAN:

BEGIN
  READ(I,K,A,X);
  READLN(S,B,Z)
END.
```

Three different but equivalent ways of providing the input are demonstrated:

1st (blanks delimit all fields)

```
23  63  'A'  -7.314E+3  'JOHN''S'  T  9.3
```

2nd (unnecessary blanks removed)

```
23 63'A'-7.314E+3'JOHN''S'T9.3
```

3rd (split over several records

```
                                        23
   63     'A'   -7314.0      'JOHN'
'S'   T   +9.3E+00
```

Formatted Input

The form e:f can be used for any type of variable. The f indicates that the required value is to lie in the next f positions from the current buffer pointer.

Numbers must be right-justified in the fields or they will be padded out with zeros. An illegal character in the field causes a run-time error message.

If the variable is of type BOOLEAN, the field of width f is scanned from left to right until the first T or F is found causing assignment of TRUE or FALSE respectively. If neither a T nor F is found in the field a run-time error message is given.

Single characters and strings are not enclosed in quotes as the field width and buffer pointer indicate the location of the string. All characters are considered valid.

The form e:f:d can be used for REAL variables only. This will cause a decimal point to be assumed d digits from the right margin of the field or if the number is specified with an exponent then d digits to the left of the E . If a decimal point is used in the number then it overrides this specification.

Example: Formatted Input

```
VAR I:INTEGER; K:SHRTINT;
    X:REAL; Z:LONG REAL;
    A:CHAR; S:STRING(10); B:BOOLEAN;
BEGIN
  READ(I:4;K:4,A:1,X:10,S:12,B:5,Z:6:1)
END.
```

If the following input record was read:

```
ØØ23ØØ1ØZØØØ-7.5391HORSE'SØHOOFXYFTZØØ7965
```

it would be equivalent to the following assignments:

```
I:=23
K:=10       zero inserted
A:="Z'
X:=-7.5391
S:='HORSE''S HOOF' truncated to  HORSE'S HO
B:=FALSE
Z:=796.5
```

4.15.3 I/O in General

The choice of format-free or formatted input and output can be made for each element in the I/O lists. The two methods can be intermixed at will in both input and output to suit the user's needs.

Example: Output

```
VAR N:INTEGER;  K:SHRTINT;  X:REAL;
    B:BOOLEAN;  C:STRING(3);
BEGIN
   K:=123;  N:=4;  X:=72.83;  B:=TRUE;  C:='ABC';
   WRITELN('0':1,K,N:3,X,X:13,X:8:2,B,B:5,B:2,C,C:2)
END
```

The following output line is produced:

```
 123  4      7.283000E 01 7.283000E 01  72.83   TRUE TRUE T  ABCAB
```

4.16 Built-In Procedures

Input/Output

These procedures are used without specifying the file names.

READ
READLN
WRITE
WRITELN

Dynamic Allocation

Tagfield values may not be specified.

NEW(X) - allocates a new variable of the type associated with X and assigns the address to the pointer variable X.

DISPOSE(X) - frees dynamic variable referenced by X.
- actually accomplishes nothing.

4.17 Built-In Functions

Boolean

EOLN - used without arguments.
- TRUE if the input pointer is currently at the end of a line; otherwise FALSE.

EOF - used without arguments.
- TRUE if end-of-file has been reached on the input file; otherwise FALSE.

ODD(X) - X must be integer.
- TRUE if X is odd; otherwise FALSE.

Arithmetic

Functions which accept an argument of type SHORT INTEGER, INTEGER, REAL or LONG REAL and return a result of the same type:

ABS(X) - absolute value

SQR(X) - square (X * X)

The following functions accept an argument of type SHORT INTEGER, INTEGER or REAL and produce a REAL result, or an argument of type LONG REAL and produce a LONG REAL result.

SIN(X)

COS(X)

ARCTAN(X)

EXP(X)

LN(X) - natural logarithm

LOG(X) - logarithm base 10

SQRT(X)

Transfer

ORD(X) - ordinal number of character or scalar constant.

CHR(X) - the character whose ordinal number is X (integer).

ROUND(X) - real value X is converted to an integer and is rounded.

TRUNC(X) - real value X is converted to an integer by truncating the fractional part.

Other

SUCC(X) - X is of type integer, user-defined scalar or subrange thereof.
- result is the successor value of X if it exists.

PRED(X) - X is of type integer, user-defined scalar or subrange thereof.
- result is the predecessor value of X if it exists.

CARD(X) - X must be a set.
- result is the cardinality (i.e., the number of members) of the set.

CPUTIME - used without arguments.
- returns the elapsed time in seconds as a real value.

4.18 Table of Standard Identifiers

Constants

FALSE, TRUE, MAXINT

Types

SHORT INTEGER (SHRTINT), INTEGER, REAL, LONG REAL,

BOOLEAN, CHAR, STRING(n) where 1<=n<=256

Procedures

READ, READLN, WRITE, WRITELN, NEW, DISPOSE

Functions

EOLN, EOF, ODD, ABS, SQR, SIN, COS, ARCTAN, EXP,

LN, LOG, SQRT, ORD, CHR, ROUND, TRUNC, SUCC, PRED,

CARD, CPUTIME

Reserved Words (in addition to the standard identifiers)

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

Alternate Symbols

Standard	Allowed
NOT	¬
OR	|
AND	&
<>	¬=

4.19 LIMITATIONS IMPOSED BY THE COMPILER

A. Limits imposed by compiler design which can not be altered

The user must bear these limits in mind when writing his programs.

1. The maximum nest of procedure and function declarations is 5.

2. 4096 Byte Limits

2.1 - The code generated for any segment (main program, procedure or function) must not exceed 4096 bytes.

- All branches are direct and all constants are stored elsewhere, so that this 4096 bytes is almost entirely code.

2.2 - A constant area is set up for each program segment. All constants, strings and address constants are stored here. This constant area for any segment may not exceed 4096 bytes.

2.3 - Each program segment has a data segment associated with it. The data segment may be of any size as long as the following fit in the first 4096 bytes.

- a register save area
- a run-time save area:
    - stores temporary results
    - FOR statement limits
    - WITH statement addresses
    - parameter lists
- storage for all simple variables
- base addresses for arrays and records

- array dope vectors
- subrange bounds
- constants declared in CONST declarations

3. Record Limitation

   3.1 - No record field may exceed 32,767 bytes in size.

   3.2 - No record field may start more than 32,767 bytes from the beginning of the record.

4. Array Limitation

   4.1 - The component size of any array must not exceed 32,767 bytes.

5. Packed arrays and records are not supported and procedures PACK and UNPACK are not implemented.

6. Only the simple forms of procedures NEW and DISPOSE are allowed. Tagfield values may not be specified.
   No garbage collection is done.

7. Only the files SYSIN and SYSPRINT are supported.

   7.1 - The program heading is not required or allowed.
   - SYSIN and SYSPRINT must be provided.

   7.2 - The only I/O routines supported are:
   - READ, READLN, WRITE, WRITELN, EOF and OLN.
   - as READ, READLN, EOF, and EOLN always refer to SYSIN while WRITE and WRITELN always refer to SYSPRINT; the file names are <u>not</u> specified.

8. Branching to global labels is not permitted.

9. Subranges of characters are not allowed.

B. Limits imposed by the default sizes of tables in the compiler

- Terminal error messages are issued for all table overflows.
- Table sizes can be modified in two ways:
  - Some table sizes can be set with a parameter on the EXEC card.
  - The table size can be modified in the PL360 procedure SETLIMIT provided with the compiler. SETLIMIT can then be re-compiled with the old version of the PASCAL compiler included in the link-edit step to create a new PASCAL compiler with increased default table sizes.

4.20 SYNTICS SYSTEM TERMINAL ERRORS

4 Hash Bucket Overflow

- The default is 700 buckets, each of 4 bytes for a total of 2800 bytes.
- The number of buckets is the maximum number of entries which the hashing routine can make in the hash string.
- All identifiers, constants, strings and labels required a bucket entry.
- The size can be modified by using the HB parameter on the EXEC card or by changing BUCKETSIZE in SETLIMIT.

5 Hash String Overflow

- Default size is 8000 bytes.
- The hash string contains the character representation of all identifiers, labels, strings, and the internal representations of constants. Each entry is also appended with a length field and a pointer field to the symbol table.
- The size can be modified by using the HS parameter on the EXEC card or by changing STRINGSIZE in SETLIMIT.

7 End-of-File Before <Program> Found

- The parser still had not recognized a <program> when the end of the source was reached.
- Some possible causes are missing END's, close comment symbols (*/) . or closing quotes around strings (').

8. Card Index Overflow

   - If the source is extremely sparse, the table maintaining card index information for error messages may overflow. The solution to the problem is to write denser code.

9. Backtracked Past Available Text

   - If there are severe syntax errors, the parser may attempt to backtrack further than the information maintained for this purpose. Correct the errors already indicated and re-submit the program.

4.21 PASCAL COMPILER TERMINAL ERRORS

These errors are produced when a compiler restriction is exceeded. Compilation immediately ceases with the recognition of one of these conditions. If the error is caused by a restriction imposed by compiler design, the source program must be modified to conform to this configuration. If the error is caused by a compiler table overflow, three possibilities exist:

- increase the table size by passing a parameter on the EXEC card.
- modify the source program to avoid the error.
- increase the table size by modifying SETLIMIT and create a new version of the compiler.

106 Program Segment > 4096 Bytes

If run-time checking code is currently being produced for this segment, it can be suppressed with $CHECK-. The problem can also be overcome by breaking up this segment up into several smaller ones.

114 Constant Segment > 4096 Bytes

The current program segment can be broken up into smaller segments, each of which is allowed a constant area of 4096 bytes.

117 Segment Data Area Overflow

The declarations for the current segment can be broken up so that one level of declaration is nested within the other.

121 Nest of Procedures and Functions > 5

Reduce the nesting level by moving some declarations to higher levels and write the segments at the same level rather than nested one within the other.

PASCAL TABLE OVERFLOW TERMINAL ERRORS

ERROR #	TABLE	DEFAULT SIZE (BYTES)	ENTRY SIZE (BYTES)	# OF ENTRIES	EXEC CARD PARAMETER	SETLIMIT VARIABLE	COMMENTS
100	SYMBOL	12000	24	500	ST	SYML	-table block structured -contains identifiers, labels constants, parameters, type descriptions, etc.
101	TYPE	240	12	20		TYPEL	-maximum nest of type decs. (Use TYPE declarations) (same table as 112)
102	POINT	240	8 4	30 60	PJ	POINTL	-maximum # of pointer forward references -maximum # of arrays and records per segment (same table as error 110)
103	RECURSION	1080	72	15		RSTACKL	-used to process nested type declarations
104	BLOCK	200	8	25		BLOCKL	-table block structured -entries for each segment, RECORD declaration or WITH record variable
105	ESD	600	12	50	ES	ESDSTL	-# of unique procedures and functions which can be called from any segment.
107	TXT	5000				PROGLGTH	-total area for code of current segment plus entry code of global segments

PASCAL TABLE OVERFLOW TERMINAL ERRORS (cont'd)

ERROR #	TABLE	DEFAULT SIZE (BYTES)	ENTRY SIZE (BYTES)	# OF ENTRIES	EXEC CARD PARAMETERS	SETLIMIT VARIABLE	COMMENTS
108	RLD	480	8	60	RD	RLDSTL	-# of procedures and functions called from any segment. -entries-20= # of arrays and records allowed in main program.
109	LABEL	200	4	50	LS	LABELL	-maximum # of labels known at any time.
110	JUMP	240	4	60	PJ	POINTL	-entry for each address to be filled in for IF, FOR, WHILE and REPEAT stats. (same table as error 102)
111	CHAIN	80	4	20		CHAINL	-maximum nest allowed for CASE statements. (same table as error 101)
112	CASE	240	16	15		TYPEL	-maximum nest allowed for CASE statements. (same table as error 101)
113	CONSTANT	5000			CL	CONSTLGTH	-total area for constants of current segment and global segments
115	PROCEDURE CALL	240	16	15		PROCL	-maximum depth to which procedure and function calls may be nested as actual parameters.

PASCAL TABLE OVERFLOW TERMINAL ERRORS (cont'd)

ERROR #	TABLE	DEFAULT SIZE (BYTES)	ENTRY SIZE (BYTES)	# OF ENTRIES	EXEC CARD PARAMETERS	SETLIMIT VARIABLE	COMMENTS
118	WITH	200	8	25		WITHL	-maximum nest allowed for WITH statements
120	FOR	80	4	20		FORL	-maximum nest allowed for FOR statements.
122	ASSIGN	80	4	20		ASSIGNL	-limits the complexity allowed for expressions
123	RUN-TIME TEMP SAVE	256				RUNSAVESIZE	-size of temporary area for WITH addresses, FOR limits, temporary results and parameter lists at run-time

4.22 EXEC CARD PARAMETERS

A. PARM. PASCAL

A.1 Standard options (default *)

* LIST - produce source listing
NOLIST - suppress source listing

* LOAD - produce object code on SYSGO
NOLOAD - suppress object code

* DECK - punch object deck
NODECK - do not punch object deck

A.2 Parameters to modify size of compiler tables

- 8 tables can be modified.
- Form: XX = n where XX represents a two-letter code indicating the table to be modified and n is the new table size in bytes.
- Codes:

ST - symbol table (terminal error 100)
- default = 500*24 = 12000 bytes (ST = 12000)

HS - hash string (terminal error 5)
- default = 8000 bytes (HS = 8000)

HB - hash buckets (terminal error 4)
- default = 700*4 = 2800 bytes (HB = 2800)

RD - RLD table (terminal error 108)
- default = 60*8 = 480 bytes (RD = 480)

CL - constant area (terminal error 113)
- default = 5000 bytes (CL = 5000)

PJ - point stack (terminal error 102)
- default = 30*8 = 240 bytes

also
- jump stack (terminal error 110)
- default = 60*4 = 240 bytes (PJ = 240)

LS - label stack (terminal error 109)
- default = 50*4 = 200 bytes (LS = 200)

ES - ESD stack (terminal error 105)
- default = 50*12 = 600 bytes (ES = 600)

A.3 Parameters to adjust several tables at once

- SMALL
  - reduces size of major compiler tables.
  - reduces core requirement for compile step by approximately 10K.
  - equivalent to:

```
ST = 6000
HS = 6000
HB = 2000
CL = 4000
PJ =  240
ES =  240
```

- BIG
  - increases size of major compiler tables.
  - increases core requirement for compile step by approximately 12K.
  - equivalent to:

```
ST = 18000
HS = 12000
HB =  4800
RD =   800
CL =  6000
PJ =   400
```

Parameters are processed in sequence from left to right, so it is possible for a parameter to modify the effect of a previous parameter.

Example:

PARM.PASCAL='HS=3000,BIG,ST=10000'

The size of the hash string would be changed to 3000 bytes. Parameter BIG would change several tables including the hash string and would actually nullify the effect of the HS parameter. Finally the symbol table size would be modified by BIG and then it would be modified again by the ST parameter. This is a useful technique as BIG would still have the required effect on several other tables.

B. PARM.GO

Normally a fixed-point overflow is considered as an error and execution is halted with an appropriate error message.

PARM.GO='MASKOFLOW'

masks the fixed-point overflow exception at run time.

4.23 $ COMMANDS IN THE SOURCE DECK

The commands begin with a $ in column 1 and may be placed at any desired place in the source deck.

Compiler defaults are indicated by * .

*	$LIST+	produce source listing
	$LIST-	suppress source listing
	$CODE+	list generated machine code
*	$CODE-	suppress listing of code
*	$CHECK+	generate run-time checking code
	$CHECK-	do not generate checking code
*	$WARN+	list warning messages
	$WARN-	suppress warnings
	$STARTC n	- indicates first significant card column - default $STARTC 1
	$ENDC n	- indicates last significant card column - default $ENDC 72
	$PAGE	continue source listing at a new page
	$TITLE	- replaces "UNIVERSITY OF MANITOBA" in page header with 40 characters starting in column 10 - continues source listing at a new page

The $ commands normally appear in the program listing. To provide the facility of using some $ commands and still be able to produce a clean source listing, two other commands are included.

*	$LIST$+	- list the $ commands - this command is always listed in the source
	$LIST$-	- suppress listing of $ commands - this command is never listed in the source.

CHAPTER 5

ORGANIZATION

This chapter demonstrates both compile-time and run-time organization. Topics covered include the run-time organization and register usage, the block structuring of the symbol table with particular reference to the hash-encoding scheme, a complete description of all possible symbol table entries, descriptions of all internal tables with their displays, and the register allocation mechanism in relation to the operand stack

The run-time organization is presented first because the information is essential to understand the operation of some of the compiler tables.

5.1 RUN-TIME ORGANIZATION

The run-time organization is determined primarily by two considerations. The first is to make the accessing of global variables as efficient as that for local variables. The second is the facility for recursive calls in the language. The size of the data area for each program segment is determined at compile time, but because of the possibility of recursive calls, a static data area is established only for the main program (PASCALDS). The data areas for procedures and functions are acquired dynamically on entry and released on exit.

Each program segment is restricted to 4K bytes of code so that all branches are direct. A separate constant area of up to 4K bytes is associated with each program segment. The prime purpose of this constant area is to store all constants and strings which are used in the associated program segment.

Main Program

All constants declared in CONST declarations, array and record base addresses, array dope vectors, as well as subrange upper and lower bounds, are stored directly in the main program data area (PASCALDS). The base addresses of the arrays and records are established using entries in the relocation dictionary.

Procedure or Function

All constants declared in CONST declarations, the offsets of arrays and records from the start of the data segment, array dope vectors, as well as subrange upper and lower bounds, are stored by

the compiler in the constant area associated with the program segment.

This information is needed by not only the current program segment but also by other segments for which this segment is global. This information is therefore moved into the segment data area when it is acquired at entry time. The array and record base addresses are computed at run time by using the offset from the start of the data area. Recursive calls imply that this initialization is performed for each invocation. For this reason, local arrays and subranges in recursively called segments add a considerable overhead.

Run-Time Register Usage

Three base registers are employed for every invocation of a program segment. Register 15 always points to the start of the current program segment. All branches in the code are done as displacements from this register.

The registers used to point to the segment data area and constant area depend on the nest level at which the current segment is defined. The main program is considered to be at level 0 while procedures and functions defined within the main program declarations are level 1, and procedures and functions defined within them are level 2, etc.

The register used for the data area is determined by (13 - nest level). For the main program this is 13. Thus PASCALDS always has register 13 as its base. In a similar manner, (12 - nest level) indicates the register used to point to the constant area for any segment. Registers (11 - nest level) down to 2 are work registers.

The maximum nest level allowed by the compiler is 5. For a procedure or function at this level, register 13 - 5 = 8 points to the data area while register 12 - 5 = 7 points to the constant area. This leaves registers 11 - 5 = 6 to 2 for work registers, while registers 9, 10, 11, 12 and 13 point to the data areas of all the global segments.

Example: Run-Time Register Usage

```
CONST N = 10;
TYPE INDEX = 1..N;
     VECTOR = ARRAY (INDEX) OF INTEGER;
VAR  A : VECTOR;
     I : INTEGER;
   PROCEDURE LEVEL1;
   VAR J : INTEGER;
      PROCEDURE LEVEL2;
      TYPE SUB = 1..10;
      VAR  X : ARRAY (SUB) OF INTEGER;
           K : SUB;
      BEGIN
         K := 7;
         X(K) := J + I + A(K) + N + 25
      END;
   BEGIN
      J := 23;
      LEVEL2
   END;
BEGIN
   I := 85;
   A(7) := N;
   LEVEL1
END.
```

For the main program, the constant N, the high and low bounds for INDEX, the dope vector for VECTOR and base address for A are all in PASCALDS.

For procedure LEVEL2, the high and low bounds for SUB, the dope vector for X and base address and offset information for X are stored in the constant area and are moved into the data area at entry time.

The line

X(K) := J + I + A(K) + N + 25

in LEVEL2 is discussed from the left:

X(K) - K and X are both local to LEVEL2. The base address of X, the dope vector for X and value of K, are all available from LEVEL2's data area pointed to by register 13 - 2 = 11

J - J is global to LEVEL2 and is available through register 13 - 1 = 12, which points to LEVEL1's data area

I - I is global to LEVEL2 and is available from PASCALDS which is pointed to by register 13 - 0 = 13.

A(K) - K is local to LEVEL2 and is available from the data area using register 11. The array A is global to LEVEL2, but the base address and dope vector are available in PASCALDS pointed to by register 13

N - N is global to LEVEL2 and is available in PASCALDS

25 - This constant is stored in the constant area for LEVEL2 and is available through register 12 - 2 = 10

Work registers 11 - 2 = 9 to 2 are available for expression evaluation. Register 15 is the base register for the code of LEVEL2. Registers 0 and 1 are universal work registers and are used for intermediate calculations. They are never left holding an address or value.

5.2 REGISTER USAGE

General Registers

15 - Base of current segment

14 - Return register - work register

- Base register of I/O routines

13 - Base of main program data segment (PASCALDS)

- contains save area

12, 11, 10, 9, 8 } Base of data segments for procedures and functions

13 - nest level - Base of data segment for procedure or function

12 - nest level - Base of constant area for procedure or function

11 - nest level, ., ., 2 } Work registers on stack

1 - Parameter list address, 0 - Returns function value } Universal Work Registers

Floating-Point Registers

0 - Returns function value, 2, 4, 6 } Work Registers

5.3 COMPILER ORGANIZATION

The symbol table plays the central role in the entire compiler. Approximately a third of the routines in the compiler are concerned with the construction and maintenance of this table while most other routines refer to it.

The basic operation of the symbol table has been discussed elsewhere (Fou73), but is repeated here for completeness.

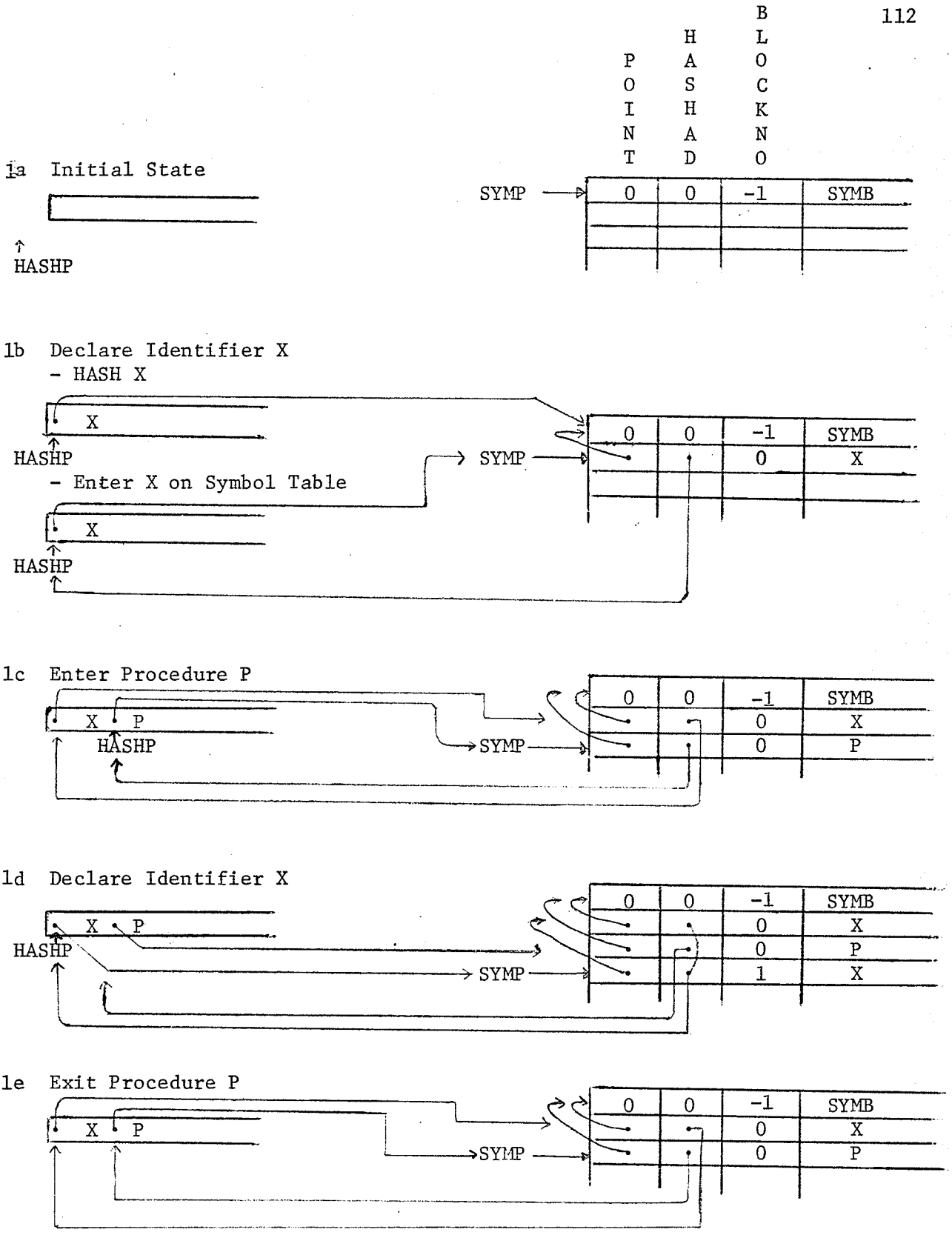
5.3.1 Hash Encoding Scheme

All identifiers, labels, constants and strings are hashed. The hashing routine returns a displacement into the HASHSTRING which then serves as the internal representation of the hashed item. Each entry in the HASHSTRING is preceded by a pointer field. This is initially set to the base address of the symbol table (SYMB), but is later updated when the item is entered on the symbol table.

Chains are maintained in the first field (POINT) of the symbol table connecting items which hashed to the same entry in HASHSTRING. The second field (HASHAD) stores the displacement of the item in HASHSTRING. The third field (BLOCKNO) gives the block nest level.

Figures 1(a) to 1(e) show the entries in HASHSTRING and on the symbol table at various times for the following program:

```
a──►  VAR X : INTEGER;
b──►    PROCEDURE P;
c──►    VAR X : INTEGER;
d──►    BEGIN
           - - -
        END;
e──►  BEGIN
        - - -
      END.
```



HASHP - pointer to the most recently hashed item in HASHSTRING

SYMP - pointer to most recent entry on the symbol table

Figures 1(a) to 1(e)

This method allows the most recent of several declarations to be chosen. If the pointer field of the hash entry points to the base of the symbol table (SYMB), then the entry has just been made by the hash routine but has not yet been entered on the symbol table.

A check to determine if an identifier has been declared before in a block is made by comparing the block number of the symbol table entry, pointed to by the pointer field in the hash entry, to the current block number. If they are the same, then the identifier is a duplicate.

This scheme is independent of the particular hashing scheme used and the method of dealing with collisions.

5.3.2 Block Control

A separate block table is used to control the block structuring of the symbol table. An entry is made on the block table whenever a new segment (procedure or function) is encountered and is removed when the segment is closed. There are two fields in each entry on the block table. The first (FENTRY) contains the symbol table pointer (SYMP) value at the time the segment is entered. The second field (FDEC) contains the value of SYMP after the parameter list has been entered on the symbol table.

Figures 1(f) to 1(j) demonstrate the mechanism for block control. Three phases of block control are discussed.

Block Entry (1f)

1) Block entry action takes place AFTER the procedure name has been entered on the symbol table. (The procedure name belongs to the outer block).

2) The block number is incremented by one and FENTRY is set equal to the symbol table pointer.

Mark Parameters (1g)

1) The field FDEC is set equal to the symbol table pointer after all (if any) of the parameters have been entered on the symbol table (1h).

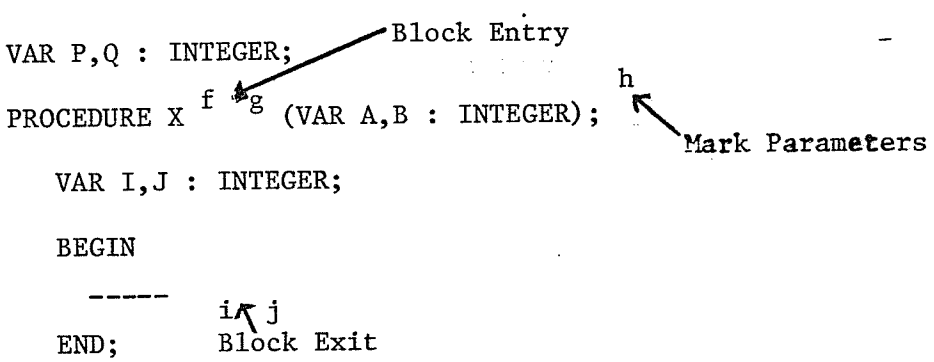
Block Exit (1i)

1) The pointer fields in the hashstring are reset from the POINT fields in the symbol table. HASHAD contains the displacements of these pointers fields in HASHSTRING and is used to calculate the correct addresses.

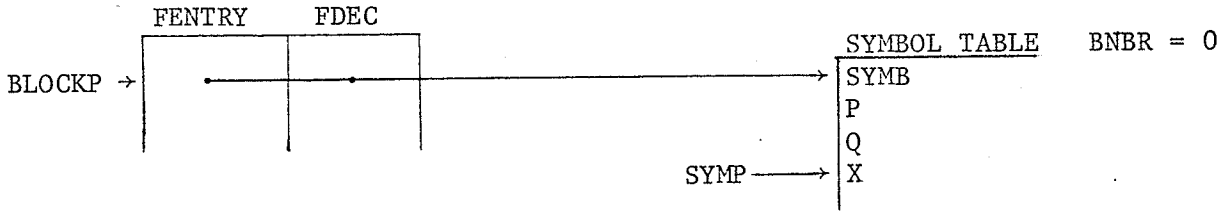
2) The block number is decremented by 1.

3) The symbol table pointer is reset to the value stored in FDEC. (Parameter information is retained so that parameter types can be checked on subsequent procedure and function calls).

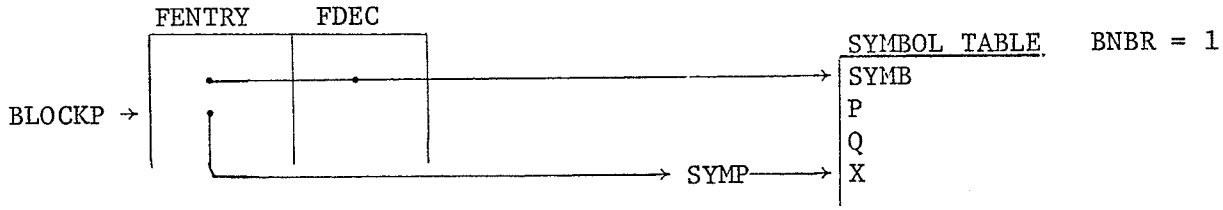
4) An entry is removed from the block table (1j).



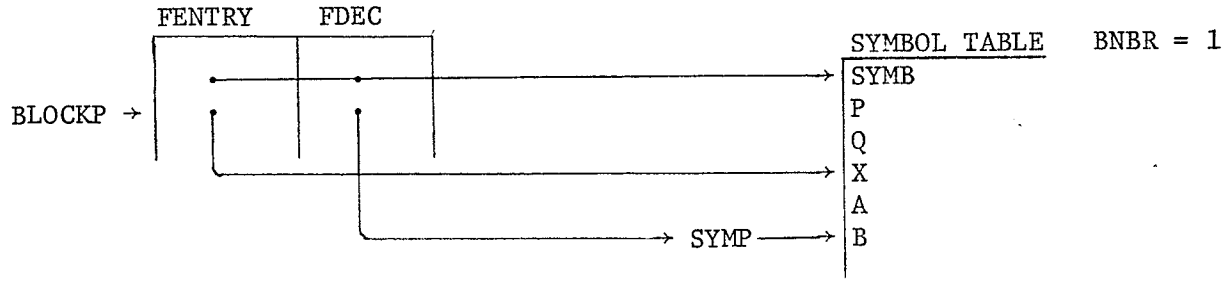
1f State Before Block Entry



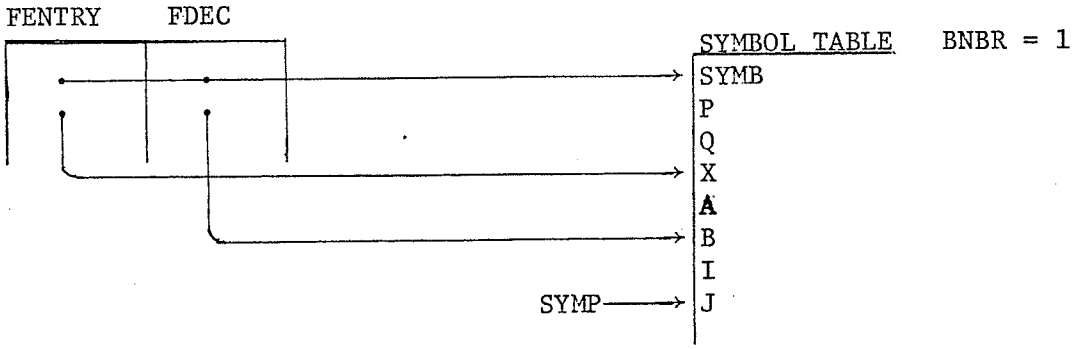
1g After Block Entry



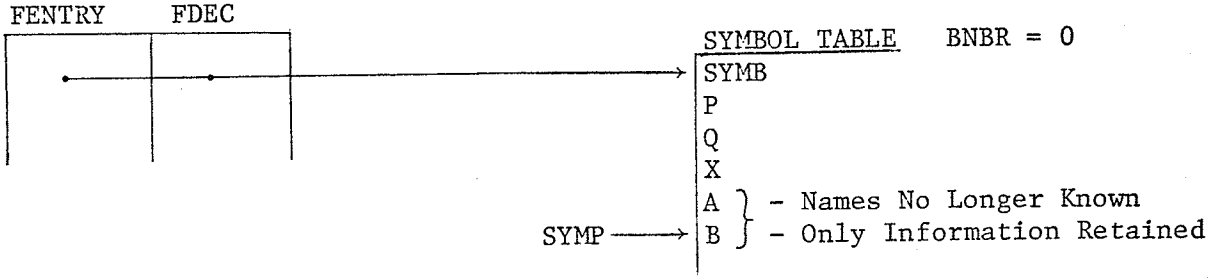
1h After Mark Parameters



1i Before Block Exit



1j After Block Exit



Figures 1(f) to 1(j)

5.4 SYMBOL / BLOCK TABLE DISPLAY

Both the symbol table and block table are controlled by one display. This display contains all the information necessary to manage the symbol table in a block structured manner. Every table in the compiler is managed through the use of displays to standardize table operation and to make modification easy to accomplish.

The SYMBOL/BLOCK table display consists of 10 integer fields.

SYMB	BLOCKB	CBLOCKB	SYMP	SYME	SYML	BLOCKP	BLOCKE	BLOCKL	BNBR

SYMB - base address of the symbol table

BLOCKB - base address of the block table

CBLOCKB - symbol table pointer indicating the base of the current block

SYMP - symbol table pointer

- address of most recent entry on the symbol table

SYME - size of symbol table entry

- in PASCAL each entry is 24 bytes

SYML - length of symbol table

(default = 1200 bytes)

- after space for the symbol table is acquired with a GETMAIN it is set to the address of the top of the symbol table so that SYMP can be checked against it to detect a symbol table overflow.

BLOCKP - block table pointer

- address of most recent entry on the block table

BLOCKE - size of block table entry (8 bytes)

BLOCKL - length of block table (200 bytes)

- after space for the block table is acquired with a GETMAIN it is set to the address of the top of the block table so that BLOCKP can be checked against it to detect a block table overflow.

BNBR - block number (initially 0)

- nest level in procedure/function declarations.

The example demonstrating block control 1f to 1j is repeated showing how the whole operation is controlled by the symbol/block table display. Initially all fields in the display are demonstrated with the further stages only showing the fields necessary to demonstrate the actions being taken.

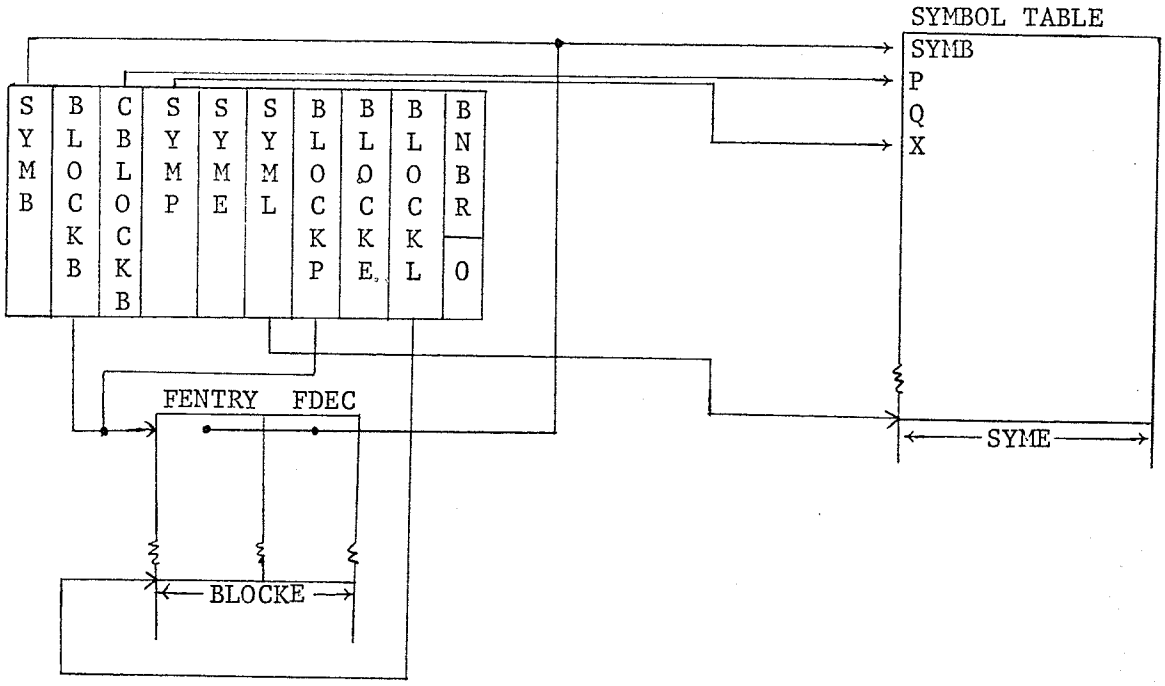
Demonstration Program:

```
VAR P,Q : INTEGER ;          <- Block Entry
PROCEDURE X   f    g   (VAR A,B : INTEGER);   h <- Mark Parameters
   VAR I,J: INTEGER;
   BEGIN
     -----
   END;        i    j
               ^
               |
          Block Exit
```

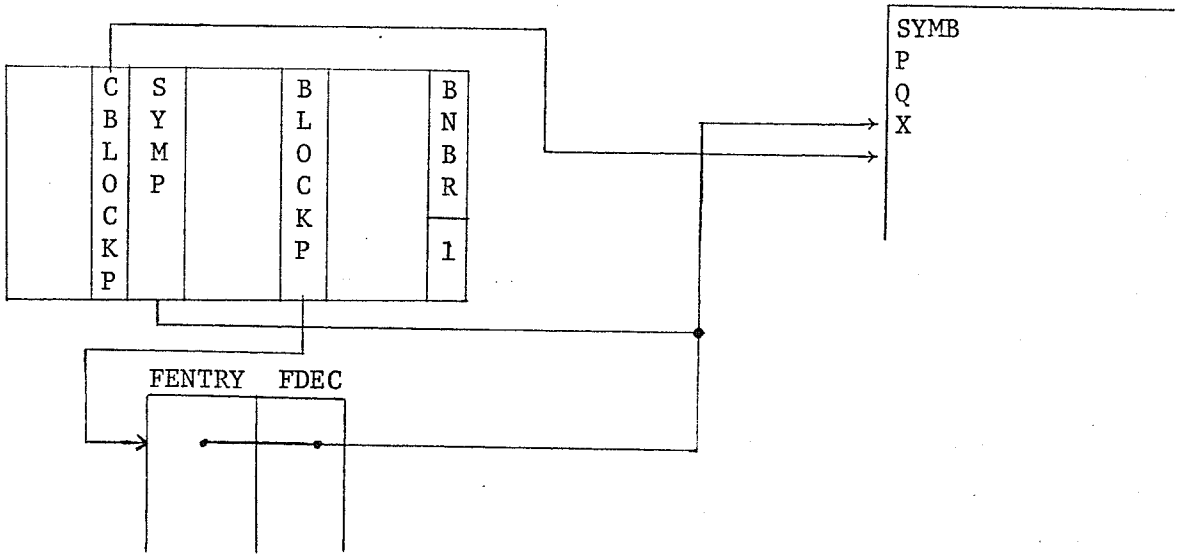
5.5 USE OF THE SYMBOL TABLE

Every identifier, label and constant used in a PASCAL program is entered on the symbol table. In addition information on subrange limits, array descriptors and record fields also resides here.

1f State Before Block Entry

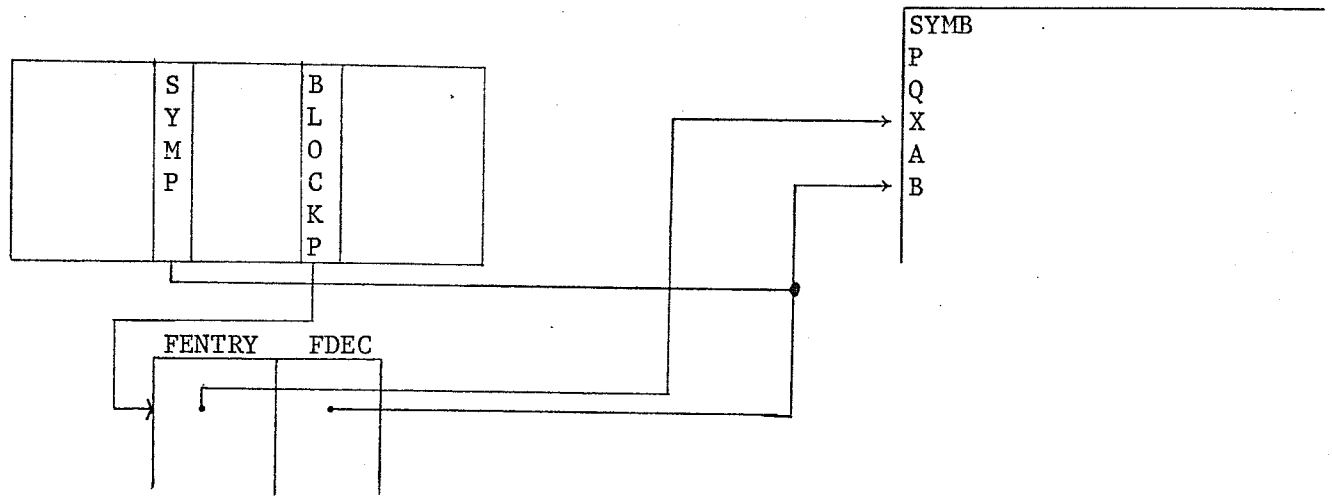


1g After Block Entry

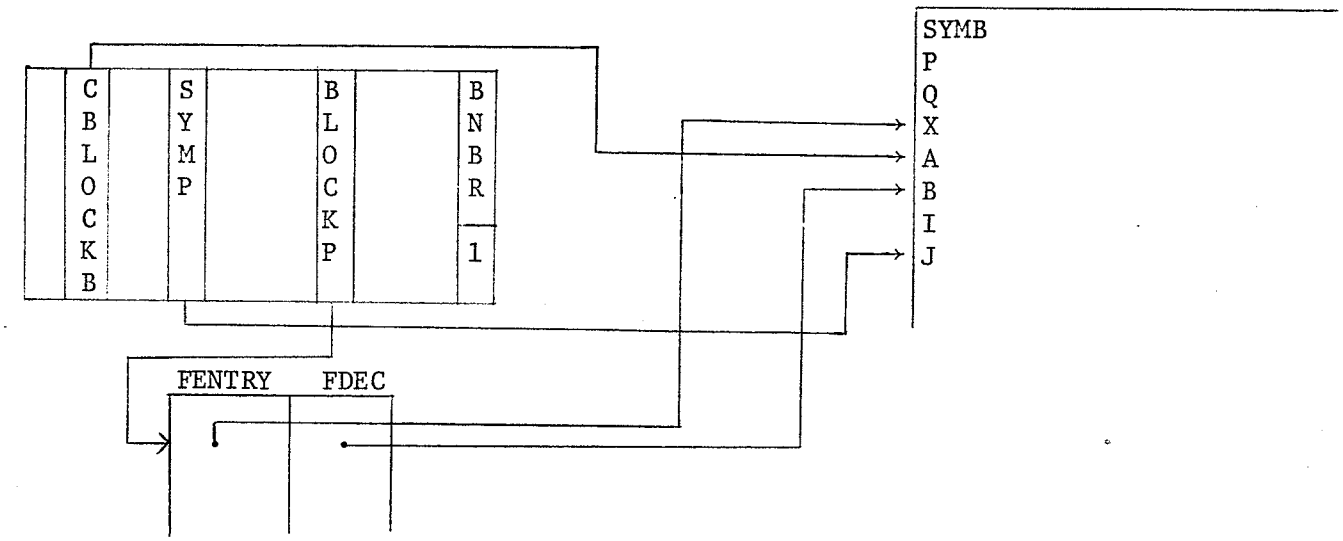


Figures 1(f) to 1(g)

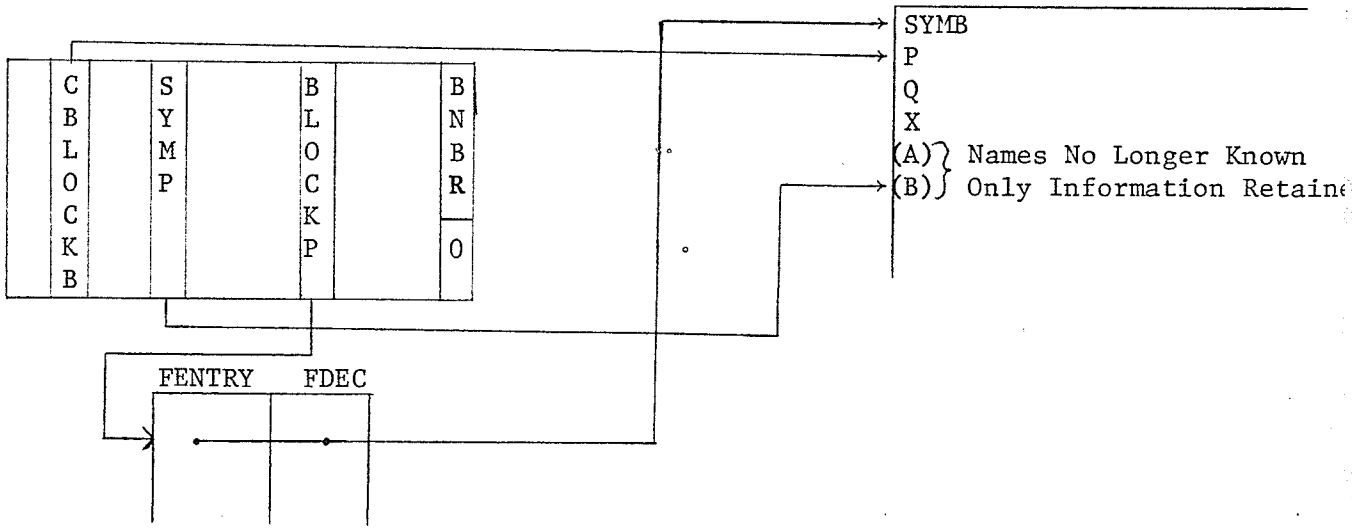
1h After Mark Parameters



1i Before Block Exit



1j After Block Exit



Figures 1(h) to 1(j)

The block structuring scheme and symbol/block table display have been described. A complete description is now given for each type of symbol table entry.

Internal Codes

All first-level identifiers (i.e., excluding record fields, etc.) are divided into 5 categories referred to as the ITYPE.

ITYPE Codes	Identifier Class
CCONST 1	constant identifier - declared in CONST declaration
CTYPE 2	type identifier - declared in TYPE declaration
CVAR 3	variable - declared in VAR declaration
CPROC 4	PROCEDURE identifier
CFUNC 5	FUNCTION identifier

In addition, the code CLABEL (0) is used for labels.

The compiler in addition has 17 type classifications which can be associated with any identifier.

Type Codes	Description
CSHRTINT 1	half-word integer
CINTEGER 2	full-word integer
CREAL 3	single precision real
CLONGREAL 4	double precision real
CBOOLEAN 5	BOOLEAN
CCHAR 6	CHAR - a single character
CSTRING 7	STRING - character string
CRADIX 8	not supported

CSET 9 set (powerset)

CSCALAR 10 user defined scalar type

CSCLEMENT 11 constant of user defined scalar

CSUBRANGE 12 subrange of some scalar type

CPOINTER 13 pointer

CARRAY 14 array

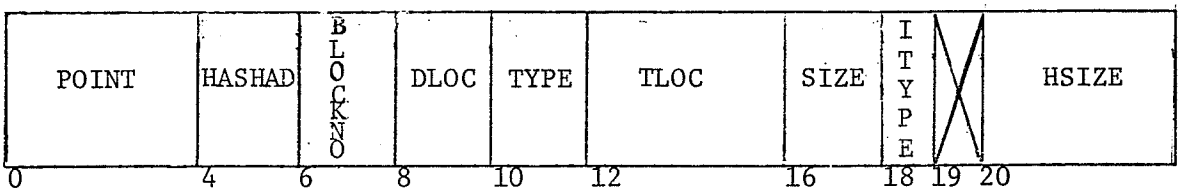
CRECORD 15 record

CFILE 16 not supported

CTYPEIDENT 17 type of variable is described by a TYPE identifier.

Every entry on the symbol table is the same size. This is specified by SYME and in the case of PASCAL is 24 bytes. These entries are put to many different uses through the use of different templates.

5.5.1 Standard Symbol Table Entry



The first 3 fields POINT, HASHAD and BLOCKNO are used to control the block structuring of the symbol table and therefore their usage is similar for most entries.

POINT - points to previous entry with same name (i.e., backchain)
- upon block exit this value is copied back into the associated pointer field in HASHSTRING restoring the old scope.

HASHAD - displacement of the associated hash entry in HASHSTRING.
- used to determine the address of the pointer field in HASHSTRING so that POINT can be copied back.

BLOCKNO - contains the value of BNBR when the entry was made (i.e. the nest level)

DLOC - contains base displacement information of the form BD where B is a register (pointing to a segment data area if the entry is an identifier or to the constant area of the current segment if the entry is a constant or string) and D is the displacement of the item from register B.

TYPE - the type associated with the identifier (one of the 17 type codes)

TLOC - pointer to the symbol table entry describing the type. This is unused for standard scalar types such as INTEGER or CHAR. The types requiring this entry are CSET, CSCALAR, CSUBRANGE, CPOINTER, CARRAY, CRECORD and CTYPEIDENT.

SIZE - the run-time core requirement in bytes for a variable of this type.

ITYPE - ITYPE code indicating the category of identifier (CTYPE, CVAR etc.)

HSIZE - the run-time core requirement in bytes for a variable of this type. This field is used in place of SIZE when the variable type is CPOINTER, CARRAY or CRECORD.

The standard entry is used for type identifiers (ITYPE is CTYPE) and variables (ITYPE is CVAR). It is used with minor modification for constant identifiers, procedure and function names (ITYPE is CCONST, CPROC and CFUNC respectively). When these standard entries are added to the symbol table all fields except DLOC and SIZE are filled in. If the identifier TYPE is a simple type, such as INTEGER or REAL, the SIZE field

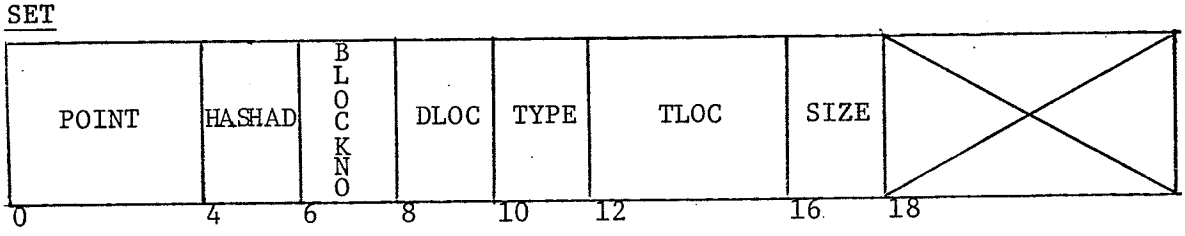
is filled in when the entry is made. DLOC and the SIZE field for structured types such as ARRAYS and RECORDS must wait until all CONST, TYPE and VAR declarations have been completed before they can be resolved.

The procedure FINDSIZE is called after the VAR declarations are complete. This procedure has two primary functions. The first is to fill in the SIZE for all type identifiers and variables which are of a structured type. The second is to allocate run-time locations in the segment data area to all variables in the segment. This location is stored in DLOC as base displacement. FINDSIZE calls a recursive procedure XSIZE to walk down chains of type declarations for structured types to determine the total run-time space required. Other functions performed by FINDSIZE and XSIZE will be discussed later.

5.5.2 Symbol Table Type Descriptors

For the standard simple types, no type descriptors are necessary and therefore TLOC is empty. These simple types are SHRTINT, INTEGER, REAL, LONGREAL, BOOLEAN, CHAR and STRING. All other data types require descriptors.

SET



HASHAD - set to 0 indicating to the block exit routine that this symbol table entry has no corresponding entry in the hashstring.

DLOC - run-time address (base displacement) of a mask for the set. The mask is a full-word and contains a 1 in every position representing a valid set element and 0's elsewhere. The mask is used by run-time checking code. The mask is located in the data area for the segment in which the set was defined.

TYPE - type of the set. There are only two possibilities. Either it is a set of some scalar (CSCALAR) or it is a SUBRANGE (CSUBRANGE) of integers or a scalar.

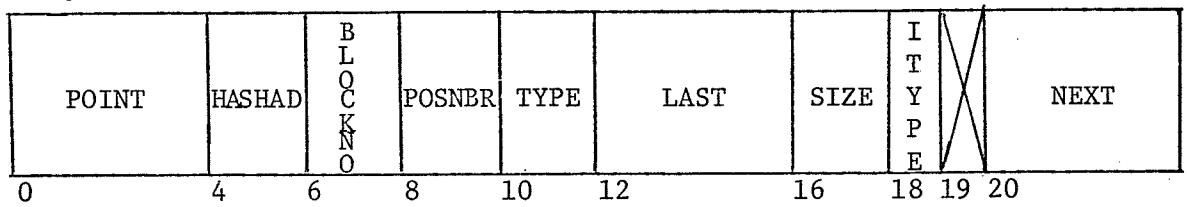
TLOC - pointer to the symbol table entry for the type description. If the TYPE is CSCALAR then it points to the first scalar constant (scelement) in the scalar. If the type is CSUBRANGE then it points to the subrange descriptor.

SIZE - a set always occupies a full-word so the size is 4.

SCALAR

A separate entry is not made for a scalar. Instead, each constant of the scalar (scelement) is entered. If a variable is of type CSCALAR then TLOC points to the entry for the first scalar constant in the list.

Entry For Scalar Constant (Scelement)



POINT, HASHAD and BLOCKNO have standard usage.

POSNBR - the position of the scalar constant in the list. The first is 0, the second 1 etc. This is the ordinal number (ORD) of the scalar constant.

TYPE - always CSCELEMENT

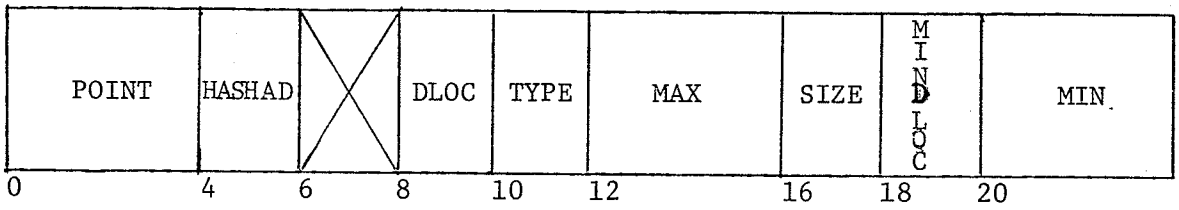
LAST - backward chain to immediately preceeding scalar constant. It is 0 for the first scalar constant

SIZE - scalar size. It is the ordinal number of the last scalar constant

ITYPE - set to CCONST because a scalar constant is used in a similar manner to a constant identifier

NEXT - forward chain to immediately following scalar constant. It is 0 for the last scalar constant

SUBRANGE



POINT - for subranges of type CSHRTINT, CINTEGER, CBOOLEAN or CSCALAR, POINT is set equal to the index size (i.e., maximum # - minimum # + 1)

HASHAD - set to 0 for block exit routine (this prevents it from trying to re-establish the scope of the item indicated by the POINT field)

DLOC - run-time address (base displacement) where the subrange upper bound is stored (i.e., the constant indicated by MAX). It is stored in the data area for the segment in which the set was defined

TYPE - the subrange type (CINTEGER, CSCALAR, etc.)

MAX - if it is a subrange of constants or constant identifiers then MAX is the address of the hashstring entry for the constant which is the upper bound. If it is a subrange of a scalar then MAX is the address of the symbol table entry for the scalar constant (scelement) which is the upper bound

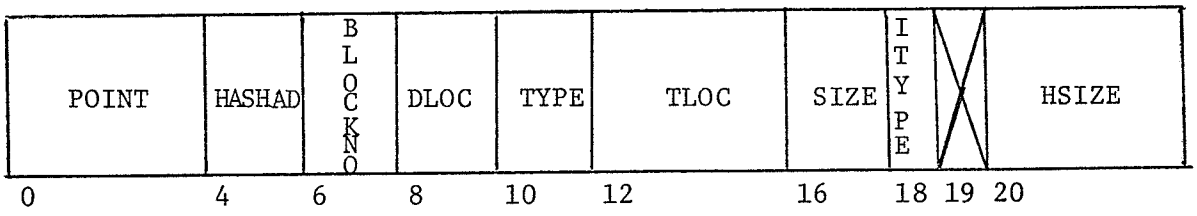
SIZE - the # of bytes needed to hold an value of the kind specified by TYPE

MINDLOC - run-time address (base displacement) where the subrange lower bound is stored (i.e., the constant indicated by MIN). It is stored in the same segment as the upper bound

MIN - if it is a subrange of constants or constant identifiers then MIN is the address of the hashstring entry for the constant which is the lower bound. If it is a subrange of a scalar then MIN is the symbol table entry for the scalar constant (scelement) which is the lower bound

POINTER

A separate entry is not made for a pointer. Instead, the necessary information is incorporated directly in the standard entry for the variable which is a POINTER.



DLOC - run-time address (base displacement) of the pointer variable

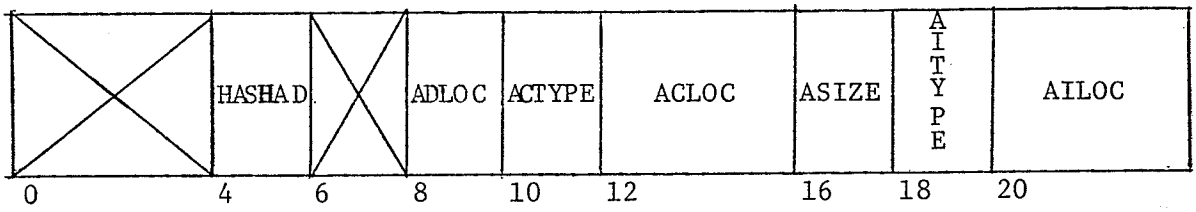
TYPE - CPOINTER

TLOC - address in the symbol table of type identifier which this variable is a pointer to

SIZE - a POINTER is an address so a full-word is allocated. The SIZE is 4 bytes

A problem arises with pointers because a variable can be designated as a pointer to a type identifier which has not yet been identified. In this case the routine FINDSIZE cannot determine the size needed to allocate an item of this type without making another pass. Instead, the size is found at code generation time by using the HSIZE field of the type identifier entry indicated by TLOC.

ARRAY



HASHAD - set to 0 so that the entry is ignored at block exit time

ADLOC - run-time address (base displacement) of the component size of the array

ACTYPE - the component type of the array (i.e., CINTEGER, CSET, CARRAY, etc.)

ACLOC - address in the symbol table of the entry describing the component type. It is empty if the component type is one of the standard scalar types (e.g. CINTEGER)

ASIZE - the component size. The size of one array component of the kind specified by ACTYPE and ACLOC

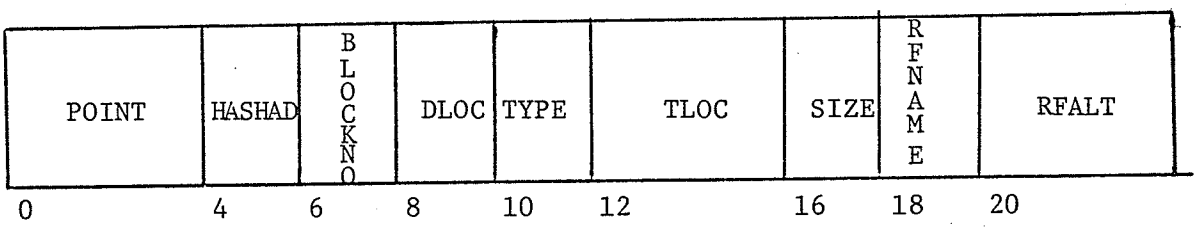
AITYPE - the index type of the array. The allowed index types are CBOOLEAN, CSCALAR and CSUBRANGE

AILOC - pointer to the symbol table of the entry describing the index type if it is CSCALAR or CSUBRANGE

RECORD

Symbol table entries for records are more complex than for other types. In particular, different entries are made for record fields, tagfields, and variant case labels.

RECORD FIELDS



POINT, BLOCKNO TYPE, TLOC and SIZE are standard

HASHAD - set to 0 for block exit routine. The record field names are not known by themselves. The displacement in HASHSTRING normally stored in HASHAD is stored in RFNAME instead.

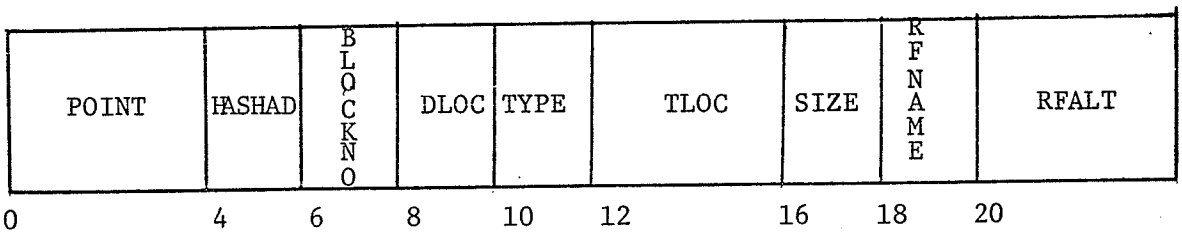
DLOC - stores the displacement of this record field from the start of the record.

RFNAME - stores the displacement of the record field name entry in HASHSTRING (normally in HASHAD).

RFALT - address of the symbol table entry of the next field in the record (0 if the last).

TAGFIELDS

If a variant part is specified without a tagfield, a dummy field of the type indicated by the type identifier is created.



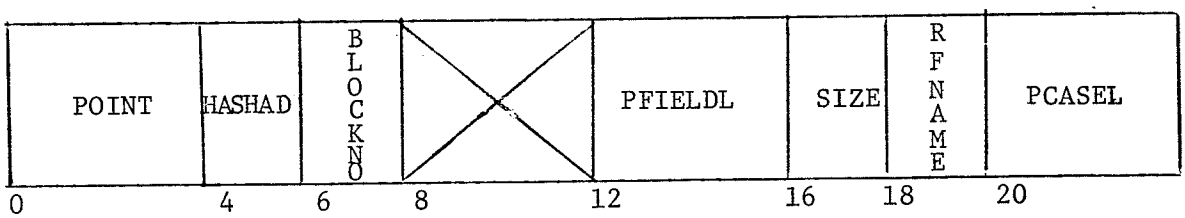
All fields except RFALT have the same use as those for record fields.

RFALT - address of the symbol table entry for the first variant case label.

The size of the largest variant is stored in the SIZE field of an extra symbol table entry immediately following the tagfield entry. This is the size used when the procedure NEW is called.

There is nothing in the tagfield entry to distinguish it from any other record field. Instead, a flag is set in the high order byte of the address field pointing to the tagfield entry. This is the RFALT field of the immediately preceeding record field if the record contains a fixed part, or the TLOC field of the record variable if the record only contains a variant part.

VARIANT CASE LABELS



POINT, HASHAD, BLOCKNO and RFNAME have the same function as those for record fields.

PFIELDL - address of the symbol table entry for the first field in that variant.

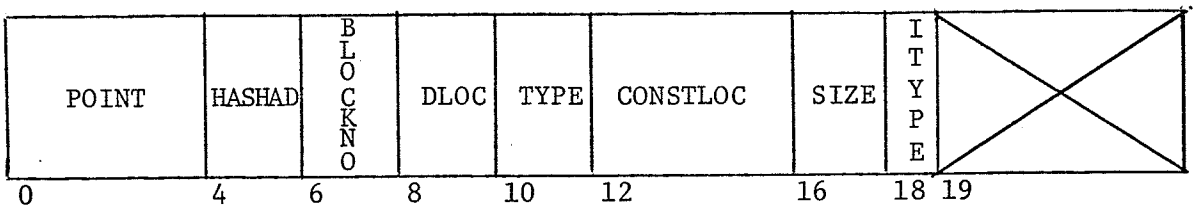
SIZE - total run-time space required for all the fields of that variant.

PCASEL - address of the symbol table entry for the next variant case label (0 if the last).

The fields for each variant are chained together in the usual manner using the RFALT fields. Each variant is considered as a complete record.

To facilitate searching through all the fields in a record when the body of a segment is being processed, XSIZE alters some of the links between entries. Intermediate links to variant case labels are removed and all record fields from the first to the last field in the last variant are linked through the RFALT field. RFALT is 0 only for the last field in the entire record description.

CONSTANT IDENTIFIERS (ITYPE is CCONST)



POINT, HASHAD, BLOCKNO, DLOC, TYPE and SIZE are standard.

CONSTLOC - absolute address of the constant (stored in the hashstring) which this constant identifier now represents.

The constant whose run-time address is stored in DLOC (base displacement) is located in PASCALDS in the main program or in the constant area for a procedure or function.

Assuming A and B are defined as constant identifiers, the declaration can have one of the following forms:

A=10; A new constant (10) is created in PASCALDS or in a procedure or function data area. The constant 10 is not entered in the symbol table. Therefore, if the constant 10 is used in the body of the segment a new constant is created in the segment constant area.

A = B; If B is defined in the main program or in the current block, then the same copy of the constant is used (DLOC, TYPE, CONSTLOC and SIZE are copied from B's symbol table entry to A's); otherwise a new constant is established in the current segment's constant area.

A = -10; The complement of the constant 10 is computed and the new constant (-10) is hashed; after which it is handled similarly to the case A = 10.

A = -B; The complement of the constant referred to by B is hashed and a new constant is created.

CONSTANTS - (used in segment bodies)

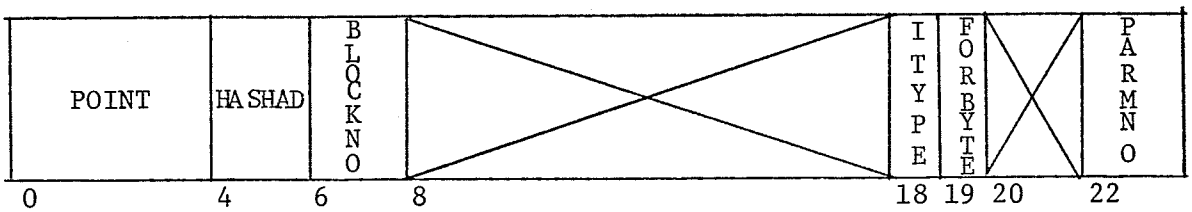
The internal representations of all constants are hashed and entries are entered on the symbol table. The constants are stored in the segment constant area. Only one copy of each constant is stored per segment no matter how often it is used.

With one exception, the symbol table entry for constants is the same as that used for constant identifiers. HASHAD now is the displacement of the hashed constant in HASHSTRING, whereas with a constant identifier it is the displacement of the hashed constant identifier.

TYPE IDENTIFIERS

When the TYPE is CTYPEIDENT and TLOC points to the symbol table entry for the type identifier, FINDSIZE eliminates this unnecessary extra link by copying TYPE, TLOC and SIZE from the type identifier entry to the other entry.

PROCEDURE

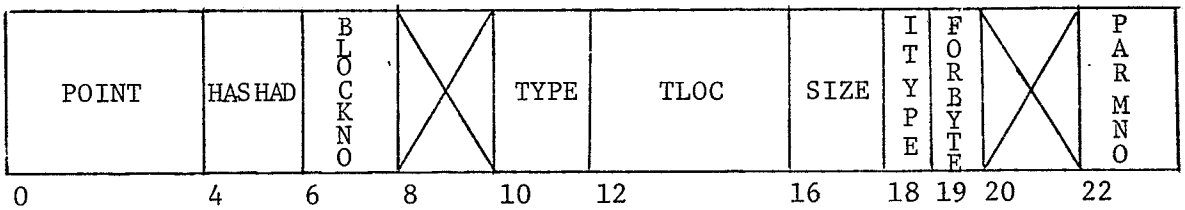


POINT, HASHAD and BLOCKNO are standard. ITYPE is CPROC.

FORBYTE - normally RESET. SET only if the procedure name is declared in a forward declaration.

PARMNO - the number of formal parameters.

FUNCTION

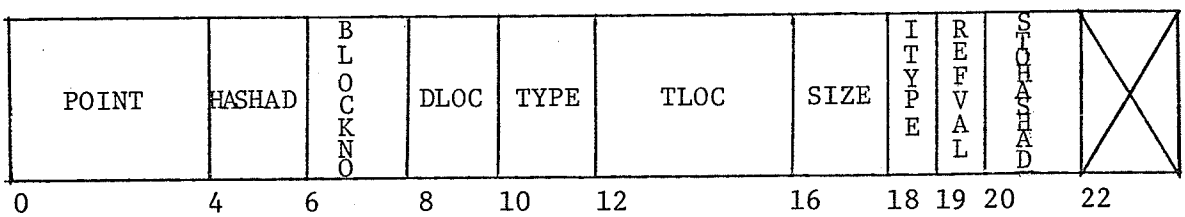


POINT, HASHAD and BLOCKNO are standard. FORBYTE and PARMNO are the same as for a procedure. ITYPE is CFUNC. TYPE, TLOC and SIZE are standard but refer to the result returned by the function.

FORMAL PARAMETERS

Formal parameters fall into 4 different categories and use 3 different templates. POINT HASHAD and BLOCKNO are standard.

VALUE AND REFERENCE PARAMETERS



DLOC - run-time address assigned to this parameter.

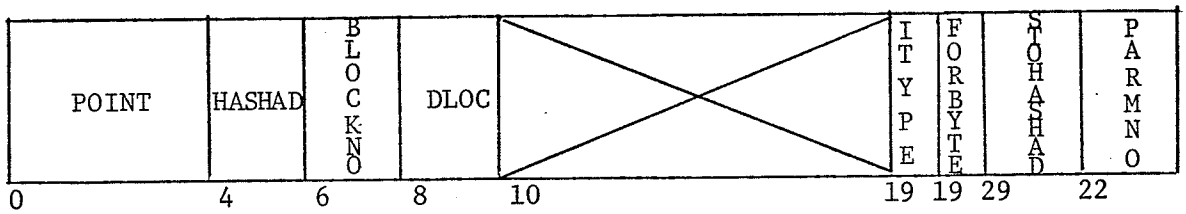
TYPE, TLOC and SIZE are standard.

ITYPE - CVAR

REFVAL - the flag is SET if it is a call by reference and RESET if it is a call by value.

STOHASHAD - this field is used only if the parameters are declared as part of a FORWARD declaration. The value in HASHAD is stored in STOHASHAD for recovery later when the procedure body is encountered.

PROCEDURE PARAMETER



DLOC - run-time location where the procedure address is stored.

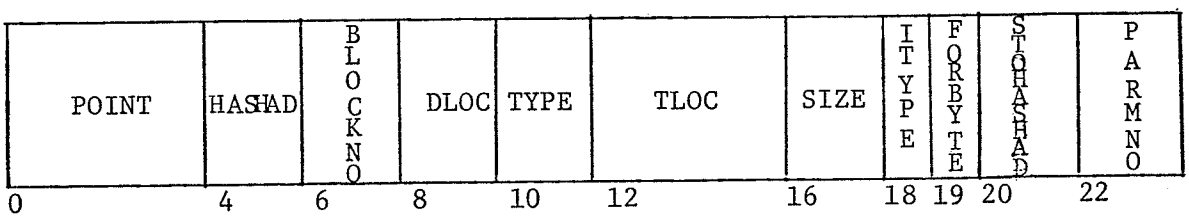
ITYPE - CPROC

FORBYTE - RESET (not announced in a FORWARD declaration)

STOHASHAD - same use as for value and reference parameters.

PARMNO - set to -1. Indicates that parameter number and type checking cannot be performed if this procedure is called.

FUNCTION PARAMETER



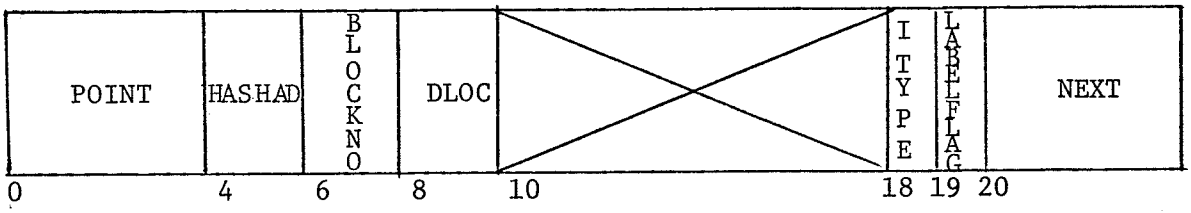
DLOC, FORBYTE, STOHASHAD and PARMNO are the same as for a procedure parameter.

ITYPE - CFUNC

TYPE, TLOC and SIZE refer to the function result type.

LABELS

The character representations of all labels are hashed and symbol table entries are made



POINT, HASHAD and BLOCKNO are standard.

DLOC - address (base displacement) of the label in the program segment.

ITYPE - CLABEL

LABELFLAG - indicates whether or not the label has been encountered in the segment. It is SET when the label is declared and RESET when the label is found

NEXT - contains label accessibility information.

5.6 INTERNAL TABLES

As demonstrated in Chapter 4, compiler table sizes may be modified by passing parameters on the EXEC card or by modifying the routine SETLIMIT. This is possible because space for all the tables is acquired at compiler-initialization time. The total memory required for all the tables is computed, a GETMAIN is issued and then the space is allocated to each table according to its specified size.

This method is feasible because all the internal tables are controlled through the use of displays. The displays vary from table to table but the basic form is the following:

Example:

Imaginary SAMPLE TABLE

SAMPLEB
SAMPLEP
SAMPLEE
SAMPLEL

The last letters of the field names (i.e., B, P, E and L) are standard for most tables. Initially SAMPLEL is the size of the table in bytes (set by SETLIMIT) and SAMPLEE is the size in bytes of a single entry in the table. At compiler-initialization time, SAMPLEL is used in the computation of the total compiler table requirements. Once space is acquired, it is allocated as follows. SAMPLEB and SAMPLEP are set equal to the address of the block of memory. SAMPLEL

is then added to this address to give the address of the top of the table and this is assigned to SAMPLEL. This same address is then used as the base address of the next table. This process continues until space is allocated to all the internal tables.

An entry is made in the table by adding the entry size (SAMPLEE) to the entry pointer (SAMPLEP) and updating the pointer. Likewise an entry is removed by subtracting the entry size from the pointer. A table overflow is detected by comparing SAMPLEP to SAMPLEL. The entries are accessed by loading the pointer value into a register and then accessing fields of the entry using a template, similar to the technique used with the symbol table.

The terms TABLE and STACK are used interchangeably and do not denote a different organization.

Table Usage Summary

ESD TABLE - stores external symbol dictionary entries for the current segment.

RLD TABLE - stores relocation dictionary entries for the current segment.

TXT AREA - stores generated code for program segments prior to creation of object modules.

CONSTANT AREA - stores constants in program constant areas prior to creation of object modules.

RUN-TIME TEMPORARY SAVE AREA - compile-time display controls use of the run-time save area.

FOR STACK - used to detect assignments to FOR statement control variables.

WITH STACK - controls accessibility of record fields on the symbol table when inside WITH statements.

CASE STACK - aids in code production for CASE statements and enables all branches to be direct.

PROCEDURE CALL STACK - controls procedure and function parameter processing including parameter number and type checking and the creation of parameter lists.

LABEL STACK - maintains pointers to label entries in the symbol table allowing checks for undefined labels at block exit.

TYPE TABLE - controls the processing of structured type descriptors.

RECURSION STACK - provides save area for simulated recursive calls.

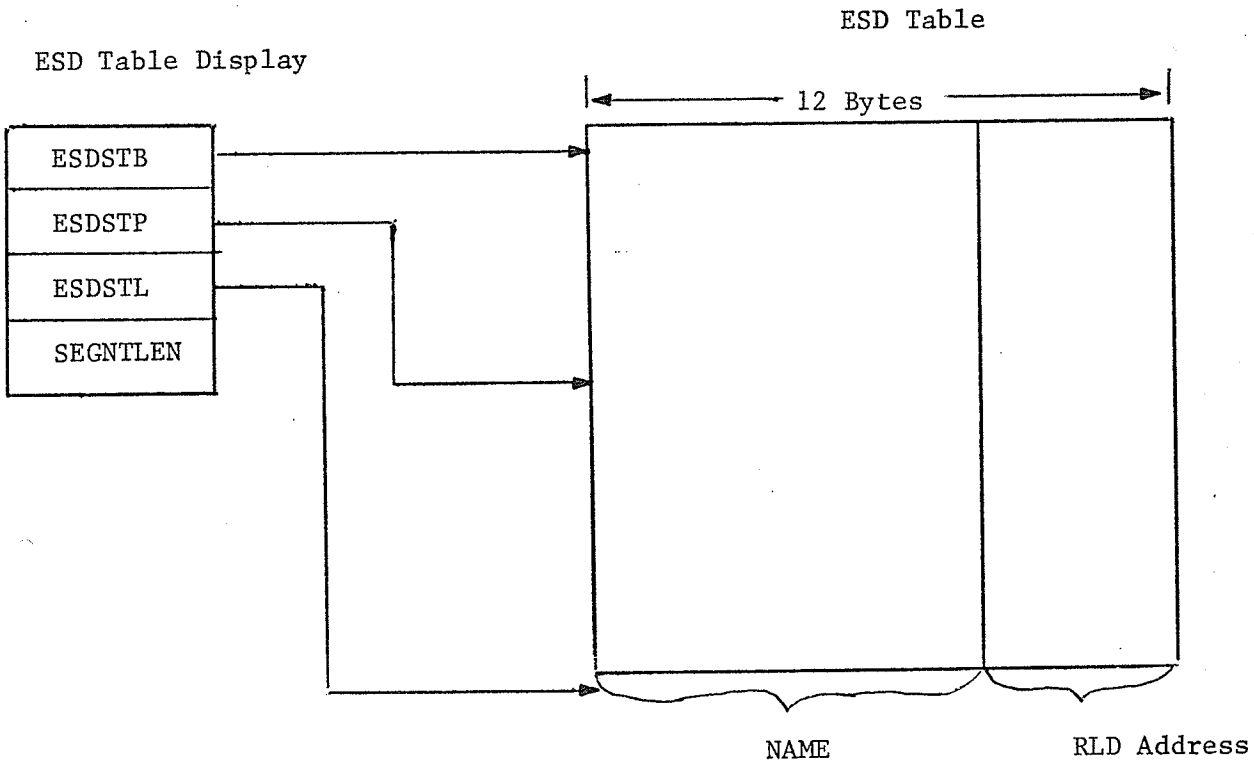
ASSIGN STACK - controls the decision as to whether values or addresses are to be used during code generation of arithmetic expressions.

POINT STACK - maintains information necessary to resolve pointer forward references.

JUMP STACK - maintains the necessary code offset information to enable all branches to be direct.

CHAIN STACK - maintains TRUE/FALSE branching information for logical expressions.

ESD TABLE



ESDSTB - base address of ESD table

ESDSTP - first free location in ESD table

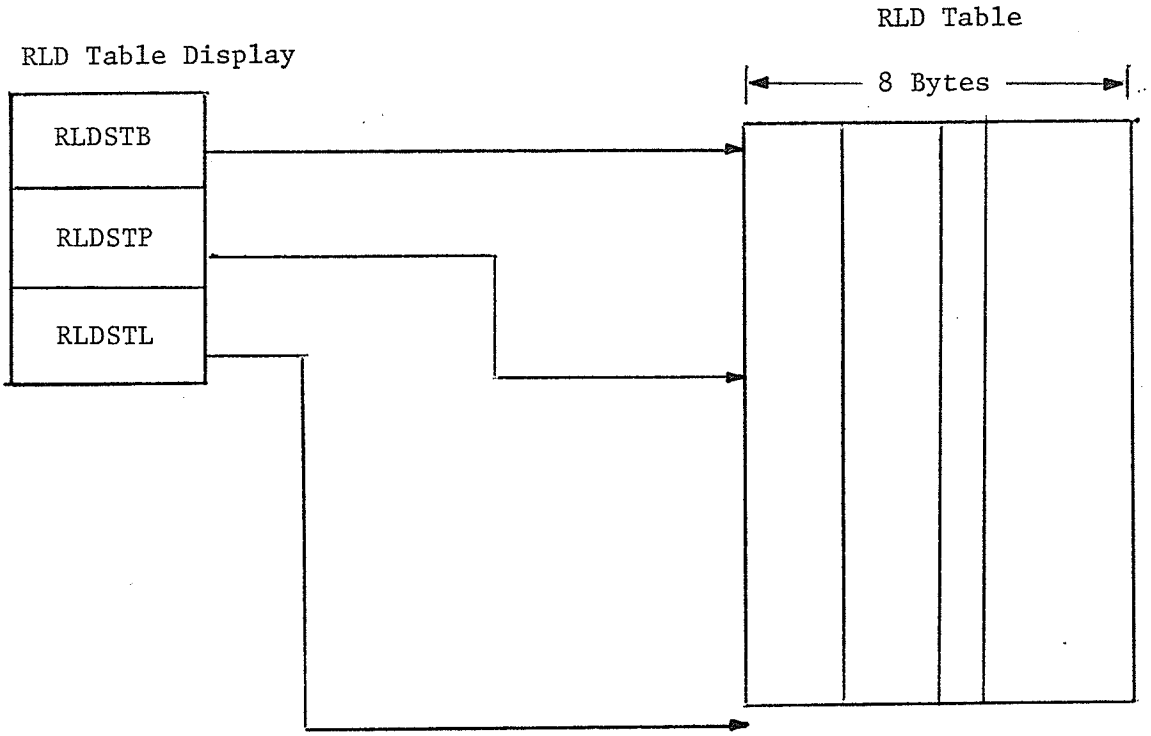
ESDSTL - initially the length of ESD table in bytes

- later the address of the end of the ESD table

SEGNTLEN - total length of the segment for which an object module is to be produced

Each entry is 12 bytes long. The name is stored in the first 8 bytes and the last 4 bytes contain the address (base displacement) of the corresponding RLD entry. Only one address constant is established for each unique ESD entry in any segment.

RLD TABLE



RLDSTB - base address of RLD table

RLDSTP - first free location in RLD table

RLDSTL - initially the length of the RLD table in bytes

- later the address of the end of the RLD table

Each entry is 8 bytes long, and is in the form of an RLD item ready to be included in the object module.

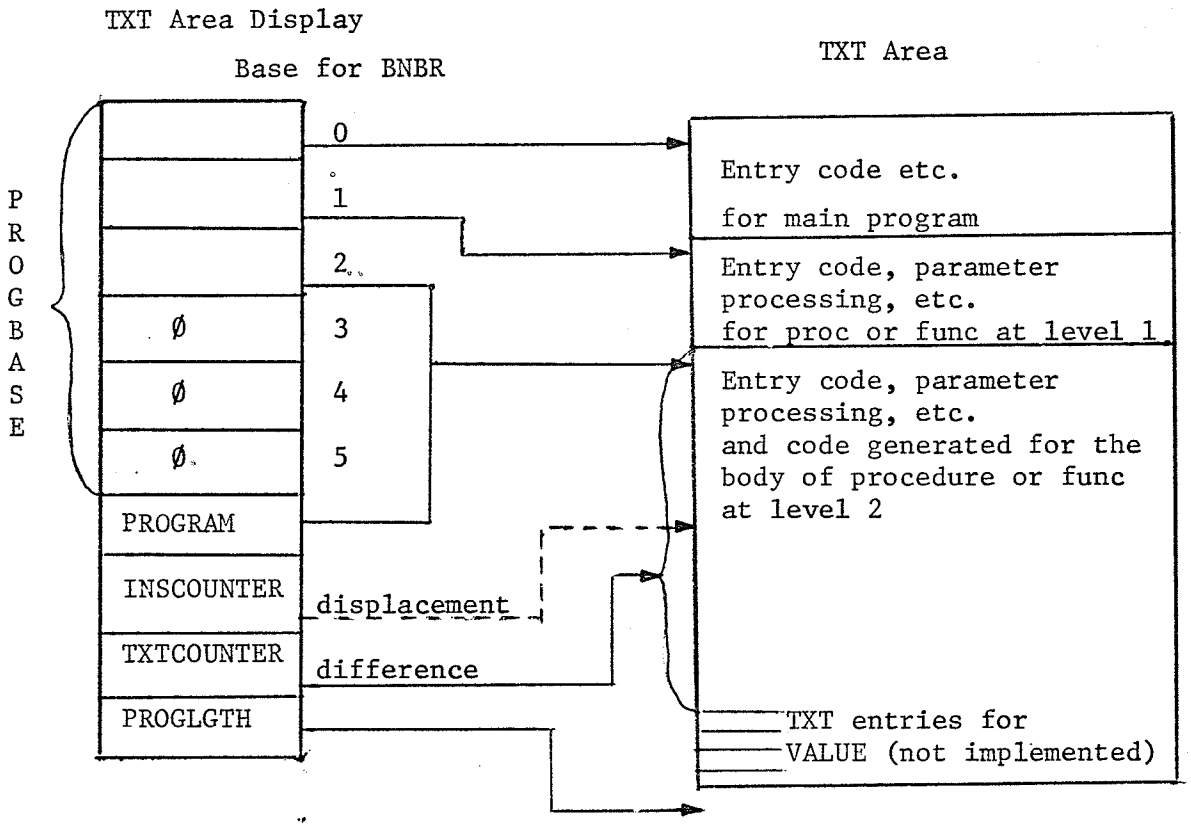
Relocation ESDID	Position ESDID	Flag (#0C)	Position Offset

RLD Entry

Bytes	1	2	3	4	5	6	-	8

TXT AREA

possible state when BNBR = 2



PROGBASE - indexed by 4 * BNBR

- base address of TXT area for each block

PROGRAM - base address of TXT area for current block

INSCOUNTER - displacement of first free location in TXT area from PROGRAM in current block

TXTCOUNTER - initially PROGLGTH - PROGRAM for any block

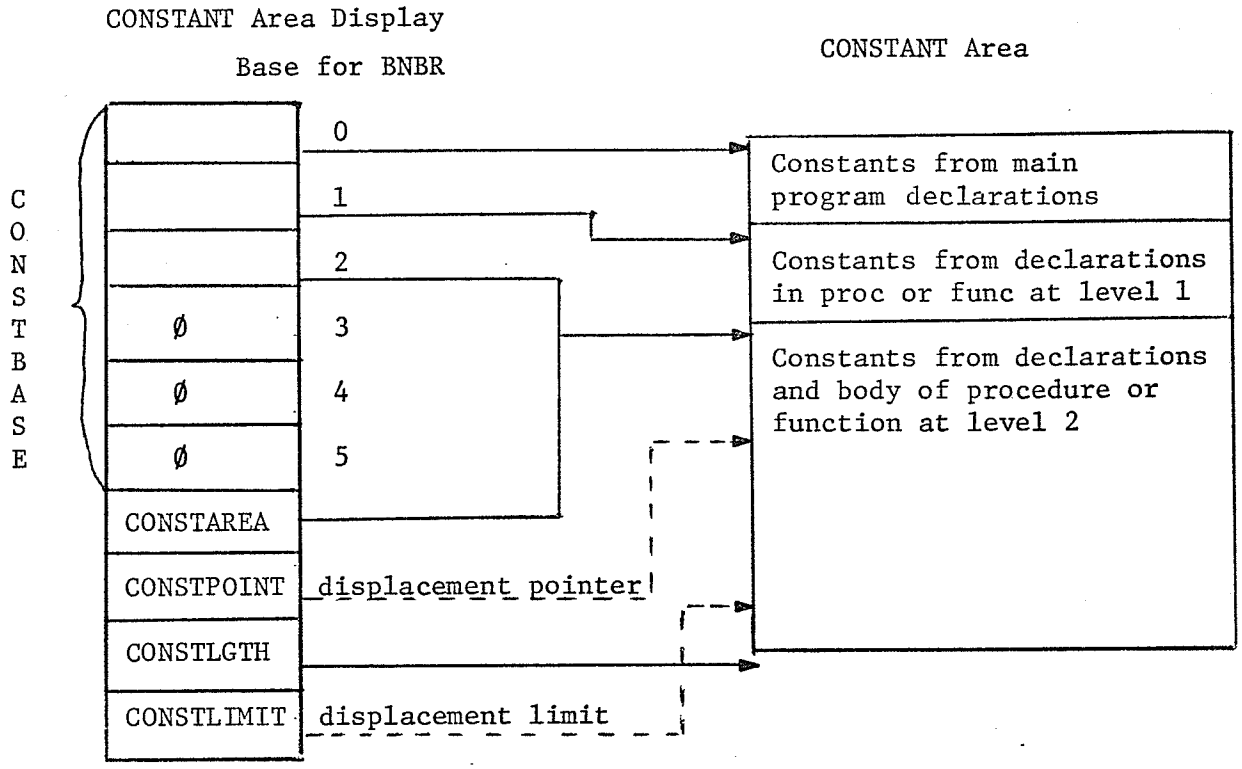
- decrements as TXT entries are added for initialization (not currently used)

PROGLGTH - initially the length of TXT area in bytes

- later the address of the end of the TXT area

CONSTANT AREA

possible state when BNBR = 2



CONSTBASE - indexed by 4 * BNBR

- base address of CONSTANT area for each block

CONSTAREA - base address of CONSTANT area for current block

CONSTPOINT - #0000RDDD where R = 12 - BNBR and DDD is the displacement of the first free location in the constant area from CONSTAREA in current block

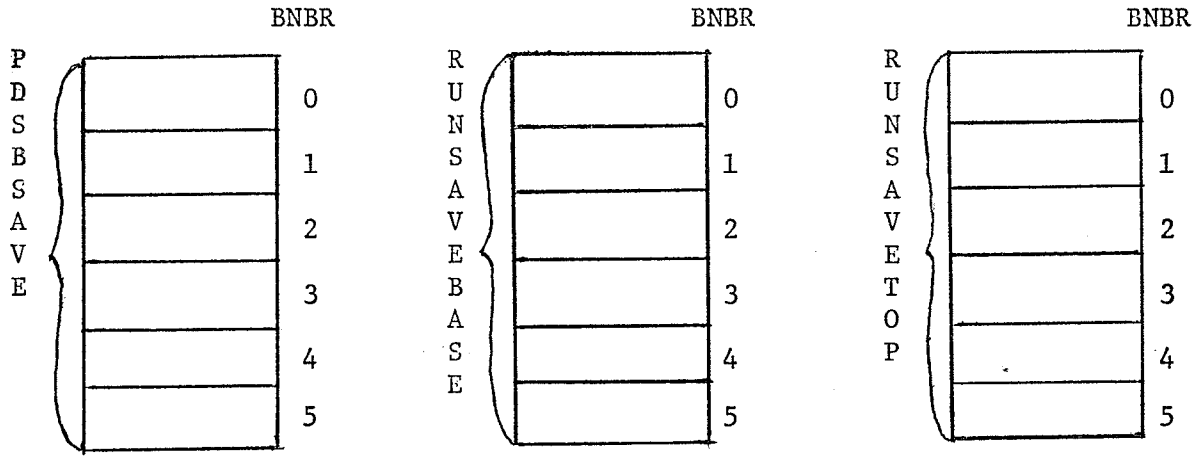
CONSTLGTH - initially the length of CONSTANT area in bytes

- later the address of the end of the constant area

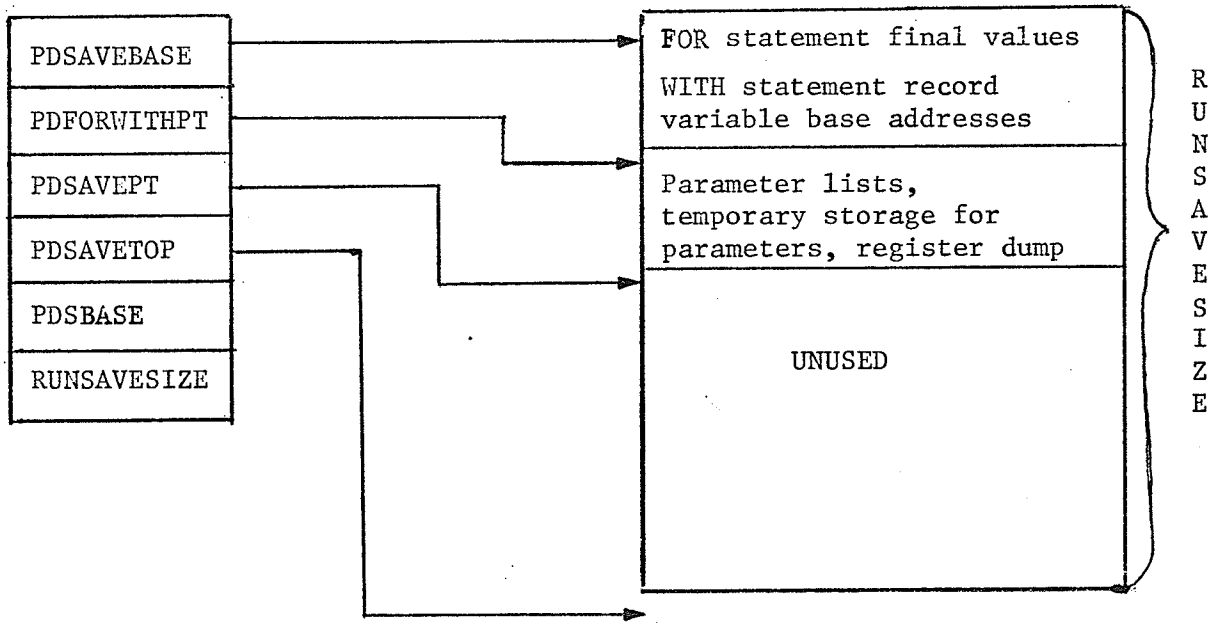
CONSTLIMIT - #0000RFFF where R = 12 - BNBR

- marks the 4K byte limit for CONSTPOINT

RUN-TIME TEMPORARY SAVE AREA



RUN-TIME SAVE AREA



PDSBSAVE - indexed by 4 * BNBR.

- stores PDSBASE for each block

RUNSAVEBASE - indexed by 4 * BNBR.

- base address of run-time save area for each block

RUNSAVETOP - indexed by 4 * BNBR

- address of the end of the run-time save area for each block

PDSAVEBASE - base address of run-time save area for current block

PDFORWITHPT - first free address in the run-time save area at which to store FOR statement final values, and WITH statement record variable base addresses.

PDSAVEPT - first free address in the run-time save area at which to store parameter lists, actual parameters and values dumped from registers

PDSAVETOP - address of the end of the current run-time save area.

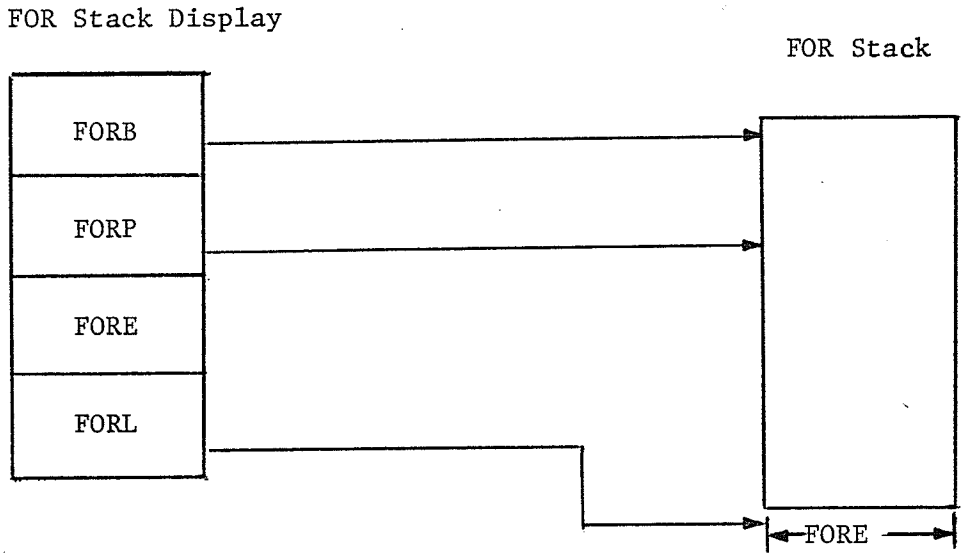
PDSBASE - #0000RDDD where R = 13 - BNBR and is the base register of the current data area, and DDD is the displacement of the first free location in the data area.

- it is this value which is assigned to the DLOC field in the symbol table for simple variables.

RUNSAVESIZE - maximum size allowed for the run-time save area. A run-time save area of maximum size (RUNSAVESIZE) is allocated in the constant area for the main program. To allow recursion, the run-time save area must be in the data area for procedures and functions. This area is allocated after the space allotted to simple variables and structured variable base addresses. The routine FINDSIZE checks to see if the structured types (arrays and records) and run-time save area will all fit into 4K. If they will then the run-time save area is allocated at the end. In this case, every time PDSAVEPT is incremented it is compared to the maximum

so far and only the exact size required for the save area is allocated at run-time. If the data area is greater than 4K then a run-time save area of maximum size is allocated before the space for the structured types.

FOR STACK



FORB - base address of FOR stack

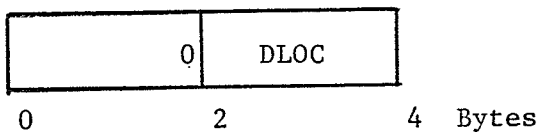
FORP - pointer to most recent entry

FORE - entry size (4 bytes)

FORL - initially the length of the FOR stack

- later the address of the end of the FOR stack

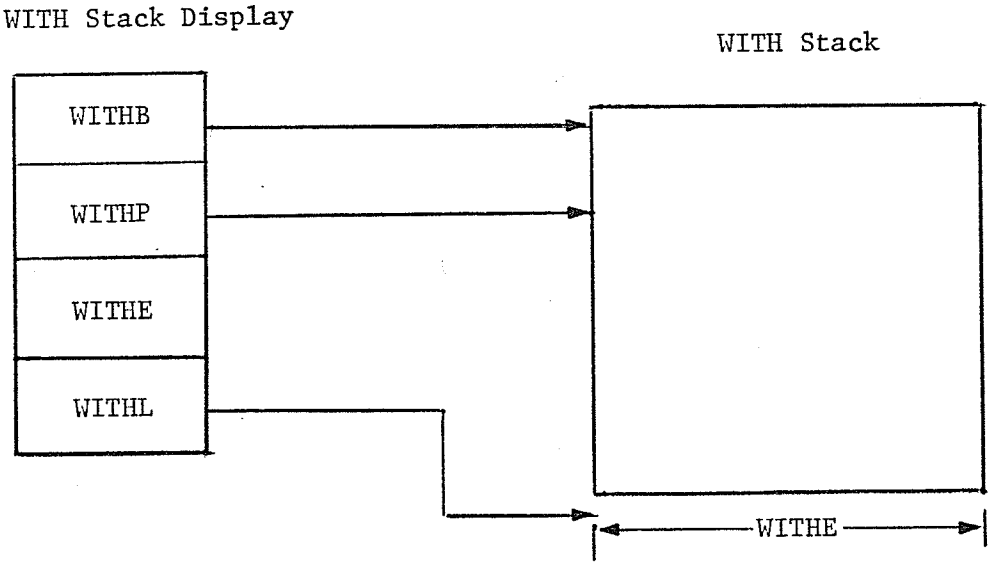
Each entry is 4 bytes long and is of the form:



where DLOC is the address (base displacement) of a FOR statement control variable.

When an assignment is made within a FOR statement and the left operand is a simple variable, a check is made to see if a FOR loop control variable is being modified by comparing the variable's address to all the entries in the FOR stack.

WITH STACK



WITHB - base address of WITH stack

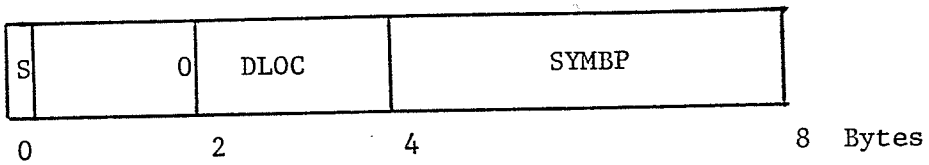
WITHP - pointer to most recent entry

WITHE - entry size (8 bytes)

WITHL - initially the length of the WITH stack

- later the address of the end of the WITH stack

Each entry is 8 bytes and is of the form



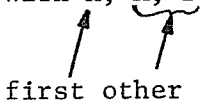
S - sign bit - 1 - first record variable in list

0 - other record variable in list

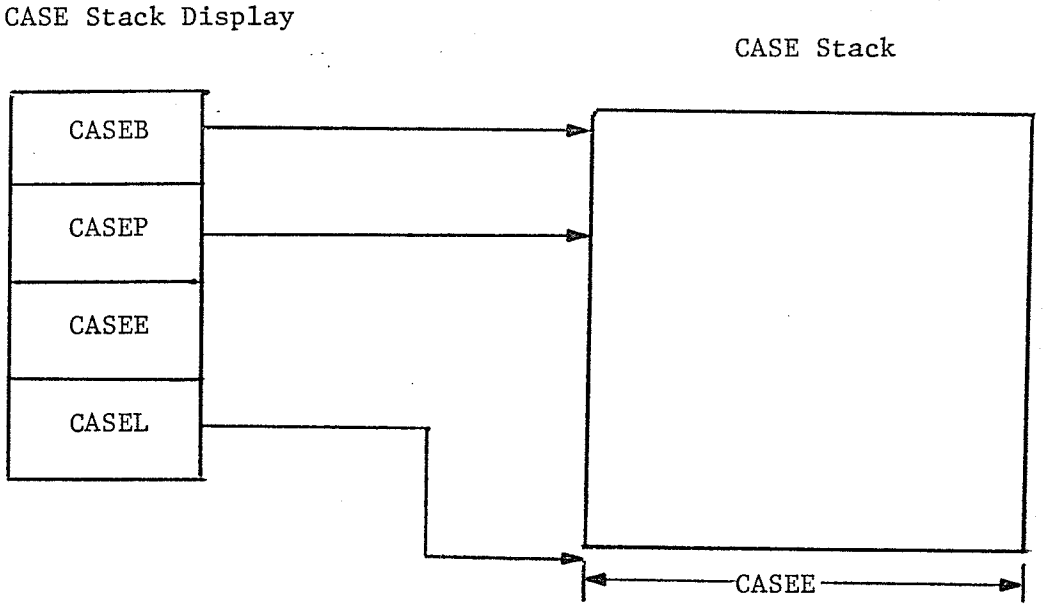
DLOC - address (base displacement) in the run-time save area containing the base address of the record variable

SYMBP - current symbol table pointer (SYMP) when the WITH statement is encountered. Allows the symbol table to be restored when the WITH is exited

e.g. WITH X, A, B DO



CASE STACK



CASEB - (TYPEB) - base address of CASE stack

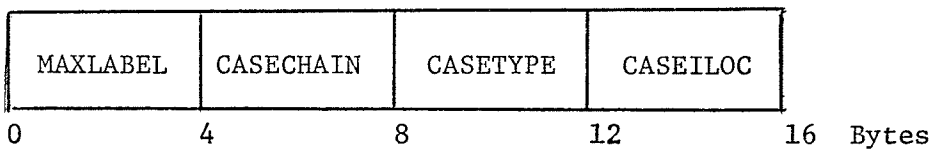
CASEP - (TYPEP) - pointer to most recent entry

CASEE - entry size (16 bytes)

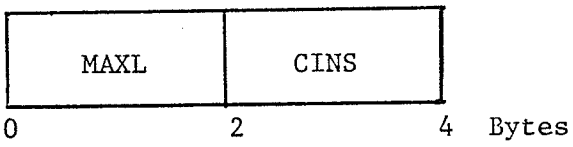
CASEL - (TYPEL) initially the length of the CASE stack

- later the address of the end of the CASE stack

An entry is made on the CASE stack when a CASE statement is encountered. Each entry is of the following form:



MAXLABEL is the address of an entry on the JUMP stack. This entry is of the form:



where MAXL is the value of the highest case label encountered so far (initially 0) and CINS is the location of the address part of a CH instruction (this address part is later set to the address of a storage location containing the maximum case label used) in the CASE selection code.

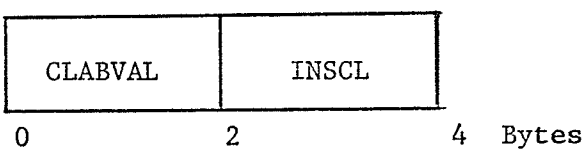
CASECHAIN is a backward chain to the address part of B NEXT instructions at the end of each alternative. Each address part indicates the previous location in the chain with the first being zero.

CASETYPE is the type (CINTEGER, CSCALAR, etc.) associated with the case selector.

CASETLOC is the address of the case selector entry on the symbol table.

Both CASETYPE and CASEILOC are used to check that case labels are compatible with the case selector.

An additional jump stack entry is created for each case label used. This is of the form:



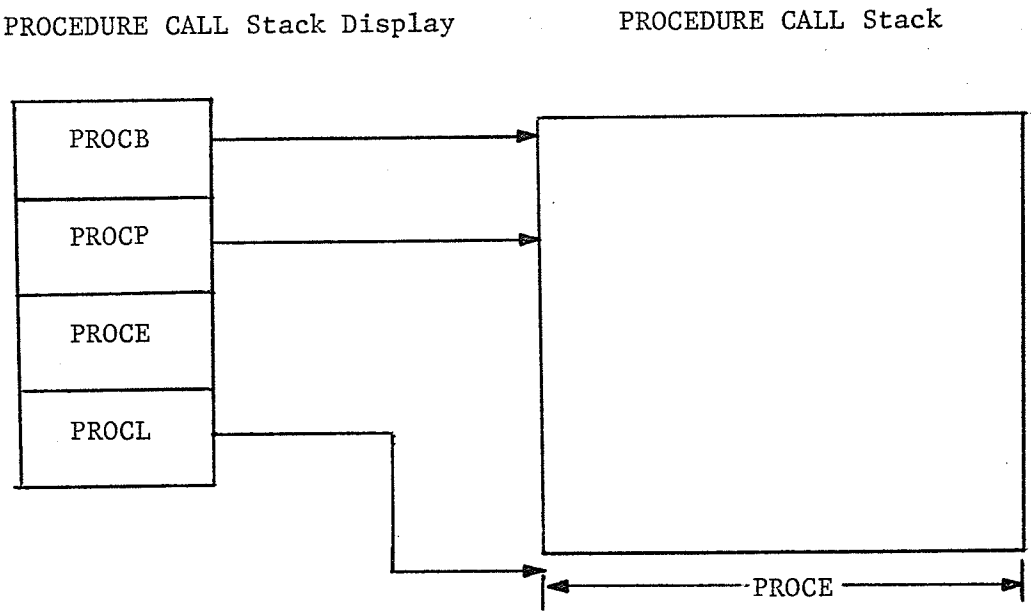
where:

CLABVAL is the label value (0-32767). This is an integer or the ordinal number of a scalar element.

INSCL is the value of INSCOUNTER (i.e., the offset in the current program segment) of the start of the code for that alternative.

The address of the branch instructions at the end of each alternative are resolved at the end of the CASE statement and a table of offsets for each alternative is constructed.

PROCEDURE CALL STACK



PROCB - base address of PROCEDURE CALL stack

PROCP - pointer to most recent entry

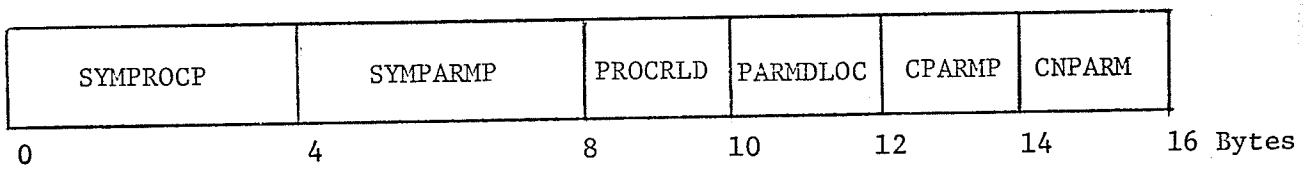
PROCE - entry size (16 bytes)

PROCL - initially the length of PROCEDURE CALL stack

- later the address of the end of the PROCEDURE CALL stack

An entry is made on the PROCEDURE CALL stack when a procedure or function call is encountered.

Each entry is of the form:



SYMPROCP - pointer to procedure or function entry on the symbol table

SYMPARMP - pointer to symbol table entry of parameter currently being processed. (If the procedure or function is a formal parameter and is not announced with a FORWARD declaration then there is no parameter checking and SYMPROCP is set to 0).

PROCRLD - address (base displacement) of the location containing the base address of the procedure or function being called.

PARMDLOC - address (base displacement) of the start of the parameter list.

CPARMP - address (base displacement) of the location (in the parameter list) where the parameter indicated by SYMPARMP is to be stored.

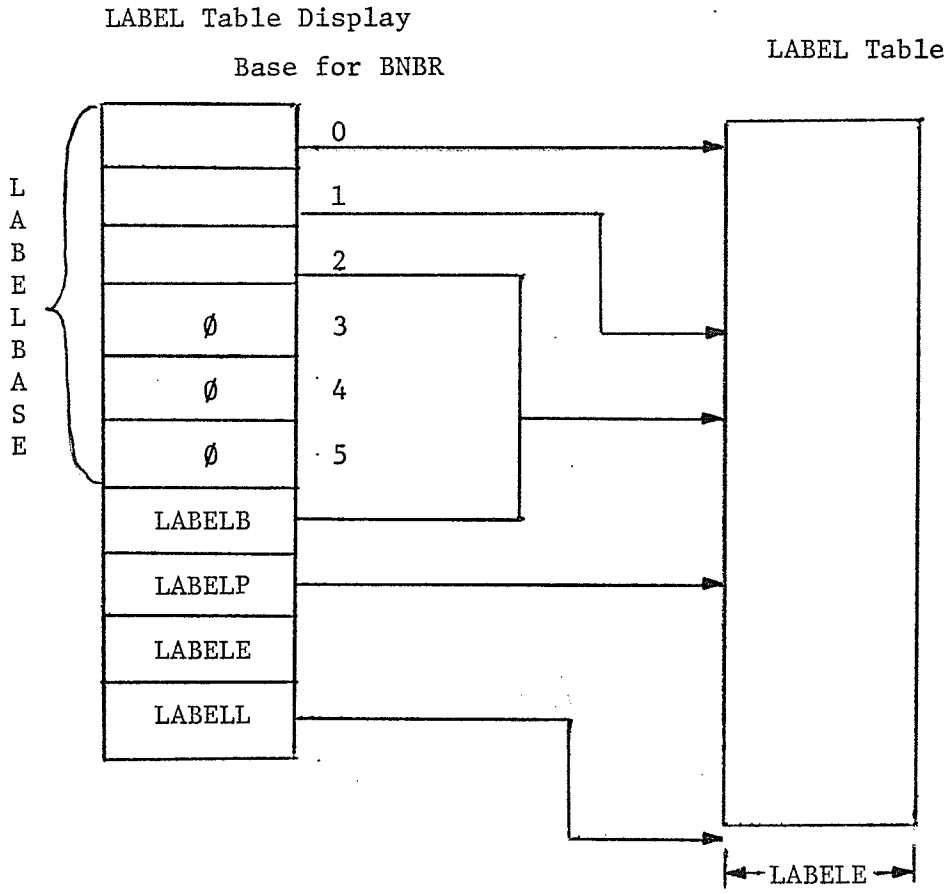
CNPARM - PARMNO (number of parameters) from symbol table.

If the number of parameters is not known (procedure or function is formal parameter and not announced by FORWARD declaration) then space is allocated for up to 10 parameters.

The parameter lists are allocated in the run-time save area starting at the location indicated by PDSAVEPT.

LABEL TABLE

possible state when BNBR = 2



LABELBASE - indexed by 4 * BNBR giving the value of LABELP when each block is entered

LABELB - base of LABEL table for current block

LABELP - current pointer

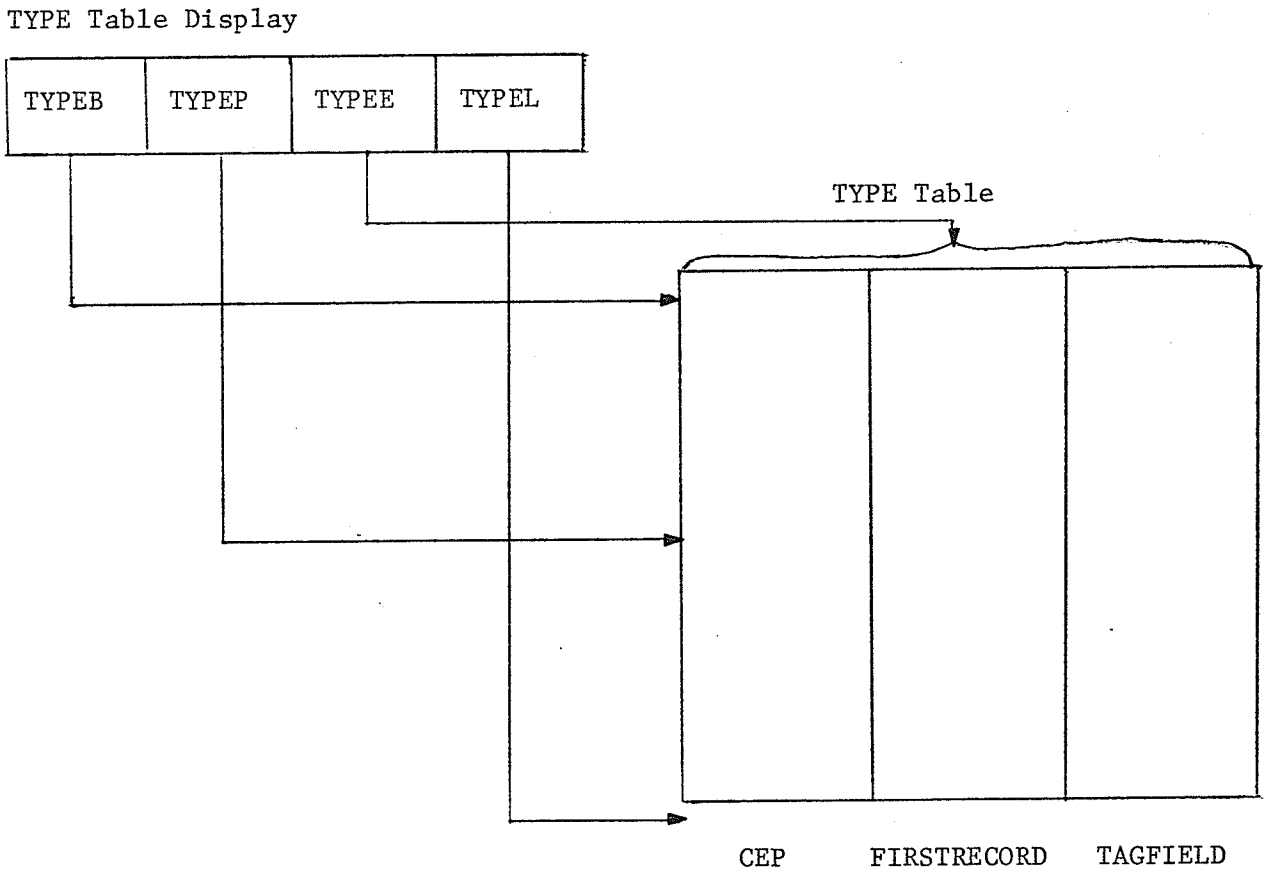
LABELE - entry size (4 bytes)

LABELL - initially the length of the LABEL table in bytes
- later the address of the end of the LABEL table

At block entry, LABELP is stored in LABELBASE (4 * BNBR) and LABELB is set equal to LABELP. At block exit LABELP is set equal to LABELB and LABELB is restored from LABELBASE (4 * BNBR).

Each label table entry is 4 bytes long and stores the address of a label entry on the symbol table.

TYPE TABLE



TYPEB — base address of TYPE table

TYPEP - pointer to most recent entry

TYPEE - entry size (12 bytes)

TYPEL - initially the length of the TYPE table

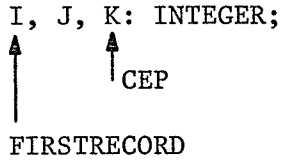
- later the address of the end of the TYPE table

CEP - current entry point in the symbol table (absolute address)

- the entry whose type is being determined

FIRSTRECORD - symbol table pointer to the first entry in a list.

e.g.



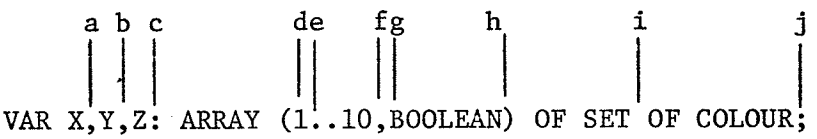
allows the type INTEGER to be stored in I through K on the symbol table.

TAGFIELD - symbol table pointer to a tagfield so that variant case labels can be checked to see that they are compatible.

Type specifications can be very long and complicated. The type table maintains the necessary pointers to the symbol table so that all the TYPE and TLOC fields can be filled in.

The operation of the type table in conjunction with the symbol table is explained by indicating the actions taken at indicated points during the processing of a single simple declaration.

TYPE COLOUR = (RED, GREEN, BLUE, YELLOW);



a - X is entered on the symbol table and an entry is made on the type table with CEP and FIRSTRECORD both pointing to X.

b - Y is entered on the symbol table.

  - CEP of the current type table entry is updated to point to Y.

c - Z is entered on the symbol table.

d - space for an array entry is acquired on the symbol table.
- the TYPE and TLOC fields of the symbol table entry (Z), indicated by CEP of the current type table entry, are set to CARRAY and the address of the array entry on the symbol table respectively.
- a new entry is made on the type table with CEP pointing to the array entry on the symbol table.

e - space for a subrange entry is acquired on the symbol table.
- the TYPE and TLOC fields of the symbol table entry indicated by CEP of the current type table entry (array entry) are set to CSUBRANGE and the address of the subrange entry respectively.
- TYPE, DLOC and MAX fields of the subrange entry are filled in.

f - DLOC, and MAX fields of the subrange entry are moved to MINDLOC and MIN, DLOC and MAX are filled in. The constants of the subrange are checked for compatibility and to ensure that the minimum value is less than the maximum value.

g - the array entry available through the CEP field of the current type table entry is checked to see that the index type specified is valid.
- the TYPE and TLOC fields are moved to AITYPE and AILOC respectively so that the component type and location can be entered.
- space for a new array entry is acquired on the symbol table.
- the TYPE and TLOC fields (component type and location of first array entry) are set to CARRAY and the address of the new array entry respectively.
- the current type table entry (pointing to first array entry) is modified (CEP field) to point to the new array entry.

h - the TYPE field of the new array entry is set to CBOOLEAN.
  - the index type is checked to see if it is valid.
  - the TYPE and TLOC fields are moved to AITYPE and AILOC respectively so that the component type and location can be entered.

i - space for a set entry is acquired on the symbol table.
  - the TYPE and TLOC fields of the second array entry (CEP from type table) are set to CSET and the address of the set entry respectively.
  - a new entry is made on the type table with CEP pointing to the set entry on the symbol table.

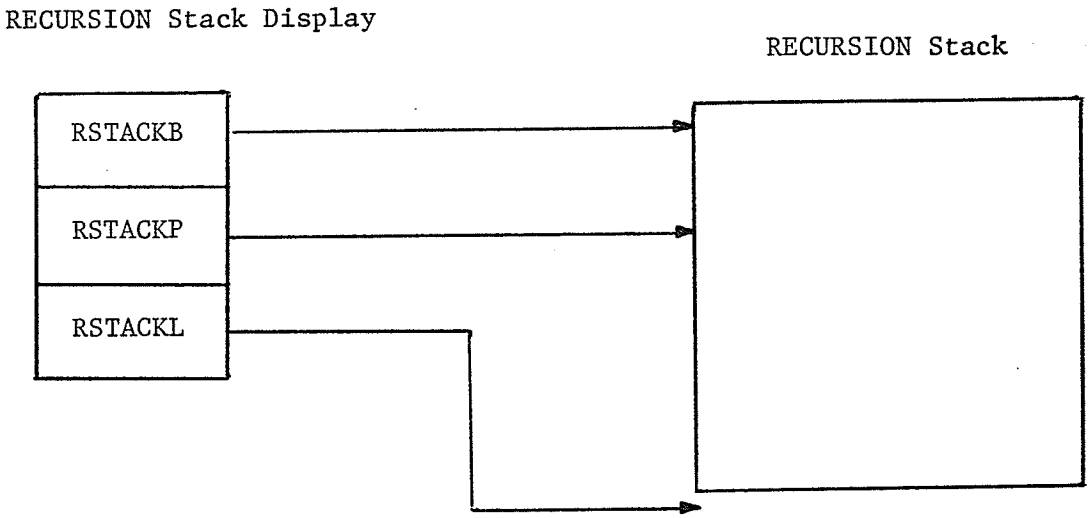
j - the TYPE and TLOC fields of the set entry are filled in as CTYPEIDENT and the symbol table location of COLOUR respectively.
  - the actual scalar is found through the TLOC field of COLOUR and is checked to ensure that POSNBR < = 31.
  - the set entry and then the array entry are removed from the type table as these types are resolved.
  - the TYPE and TLOC fields of Z are copied to those of X and Y through the use of the CEP and FIRSTRECORD fields of the first type table entry.

The SIZE and DLOC fields are resolved later when the routine FINDSIZE is called at the end of the declarations.

The use of the type table gets considerably more involved when records with variant parts are nested.

The TYPE table is required only when processing declarations. To achieve a greater utilization of this space, it is used for the CASE stack when processing segment bodies.

RECURSION STACK



RSTACKB - base address of RECURSION stack

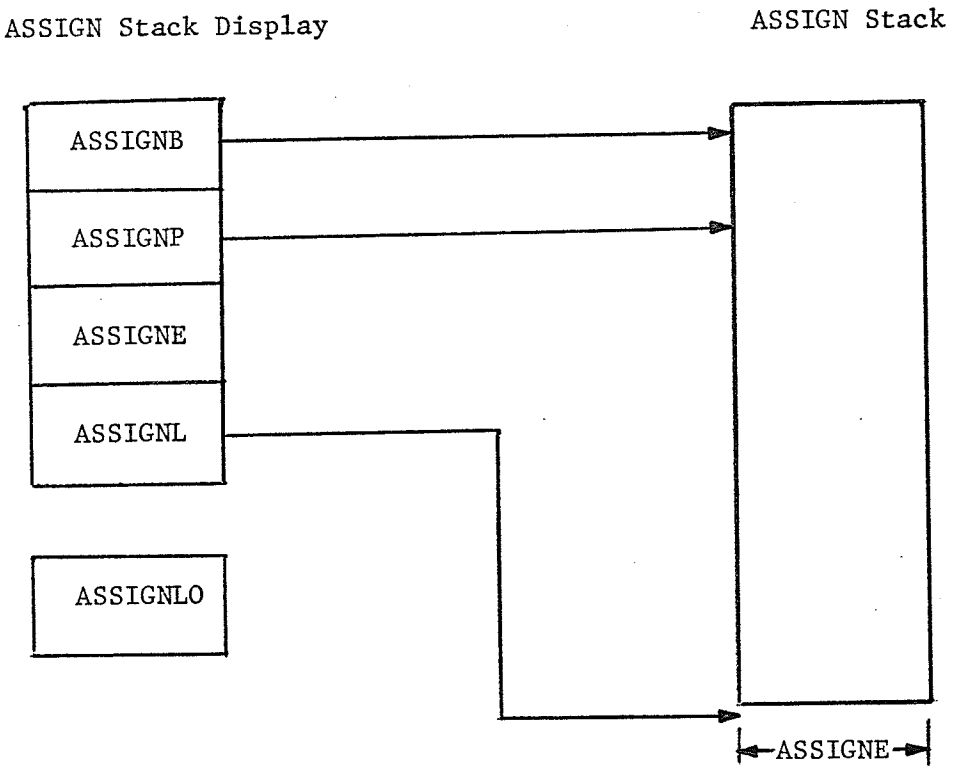
RSTACKP - address of first free location in RECURSION stack

RSTACKL - initially the length of the RECURSION stack

- later the address of the end of the RECURSION stack.

The RECURSION stack provides register save areas so that recursive calls of PL360 procedures can be simulated, as this feature is not supported in the language. It is used only by XSIZE in processing declarations of structured types.

ASSIGN STACK



ASSIGNB - base address of ASSIGN stack

ASSIGNP - pointer to the most recent entry

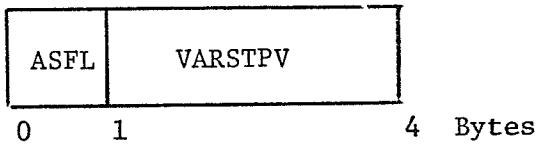
ASSIGNE - entry size (4 bytes)

ASSIGNL - initially the length of the ASSIGN stack

- later the address of the end of the ASSIGN stack

ASSIGNL0 - byte flag controlling whether or not values are to be loaded into registers.

Each entry on the ASSIGN stack is of the form



where ASFL is the current value of ASSIGNLO and VARSTPV is the current value of VARSTP (the operand stack pointer) when the entry is made.

Usually when operands such as A(I) or X.B are encountered, the address of the array element or record field is computed and the corresponding value is loaded into a register. In certain circumstances it is the address rather than the value which is of interest. An entry is stacked when a procedure or function call is encountered and is removed when the parameter list has been processed.

Examples: Assume A and B are arrays:

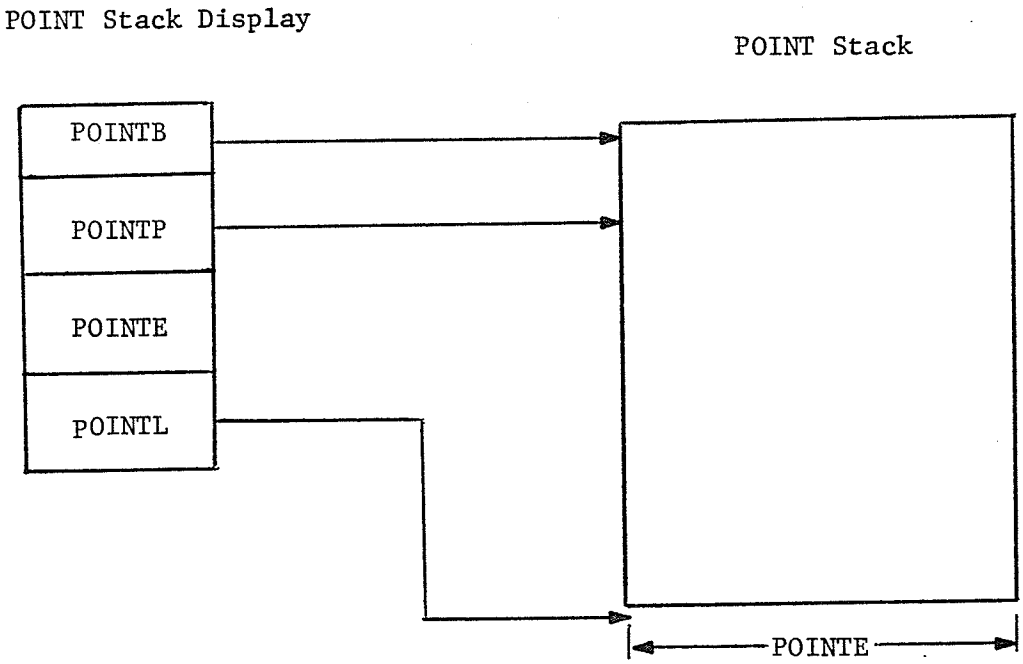
```
READLN (A(B(I)));
          value
        address

WRITELN (A(B(I)));
           value
         value
```

If SUM is a function and X is a record

```
A(B(SUM(X.B))):= X.B;
       address   value
     value
   value
 address
```

The current setting of ASSIGNLO and the difference between VARSTP and the current VARSTPV control the address/value decisions.

POINT STACK



POINTB - base address of POINT stack

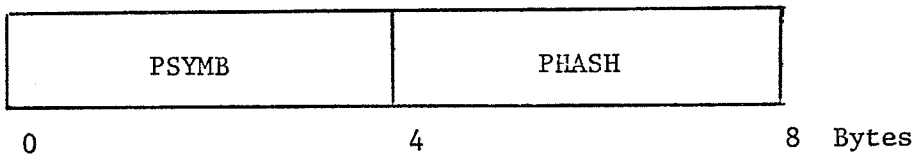
POINTP - pointer to most recent entry

POINTE - entry size (8 bytes).

POINTL - initially the length of the POINT stack

- later the address of the end of the POINT stack

Each entry is 8 bytes and is of the form:



where

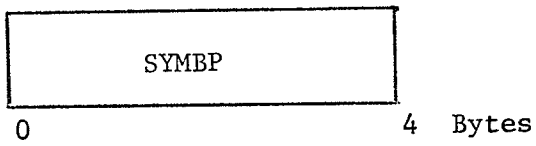
PSYMB - is the address of the symbol table entry which is declared as a pointer,

PHASH - is the address of the type identifier in HASHSTRING to which the variable indicated by PSYMB is pointing.

Entries of this form are only made on the POINT stack if the type identifier is not yet known (i.e., a forward reference). These links are resolved at the end of the TYPE declarations by checking to see if the identifier indicated by PHASH has been defined and is a TYPE identifier.

Once the type declarations are complete, the POINT stack in the above form is no longer used.

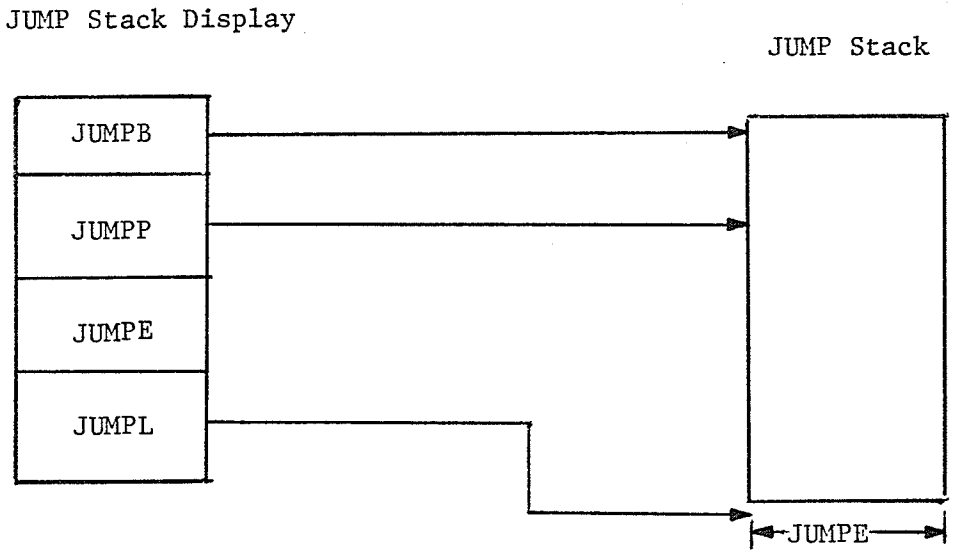
The routine FINDSIZE uses the POINT stack with entries of the form:



where SYMBP is the address of a symbol table entry for an array or record. Space is allocated first for all simple variables. This use of the POINT stack allows space to then be allocated to the structured variables without having to search through all the symbol table entries.

The POINT stack is required only when processing declarations. To achieve a greater utilization of this space, it is used for the JUMP stack when processing segment bodies.

JUMP STACK



JUMPB - (POINTB) - base address of JUMP stack

JUMPP - (POINTP) - pointer to most recent entry

JUMPE - entry size (4 bytes)

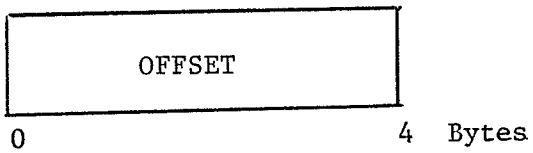
JUMPL - (POINTL) - initially the length of the JUMP stack

- later the address of the end of the JUMP stack

The JUMP stack is used to enable all branches in the object code to be direct. It is used in generating code for statements such as: IF, FOR, WHILE, REPEAT and CASE, although the way the stack is used depends upon the particular circumstances. Statements FOR, WHILE and REPEAT require backward branches while IF, FOR, WHILE, and CASE require forward branches.

Backward Branches

A statement like WHILE or FOR requires control to be passed back to the start for a test. The address which must later be used in a branch instruction must therefore be retained until the end of the statement. An entry of the form



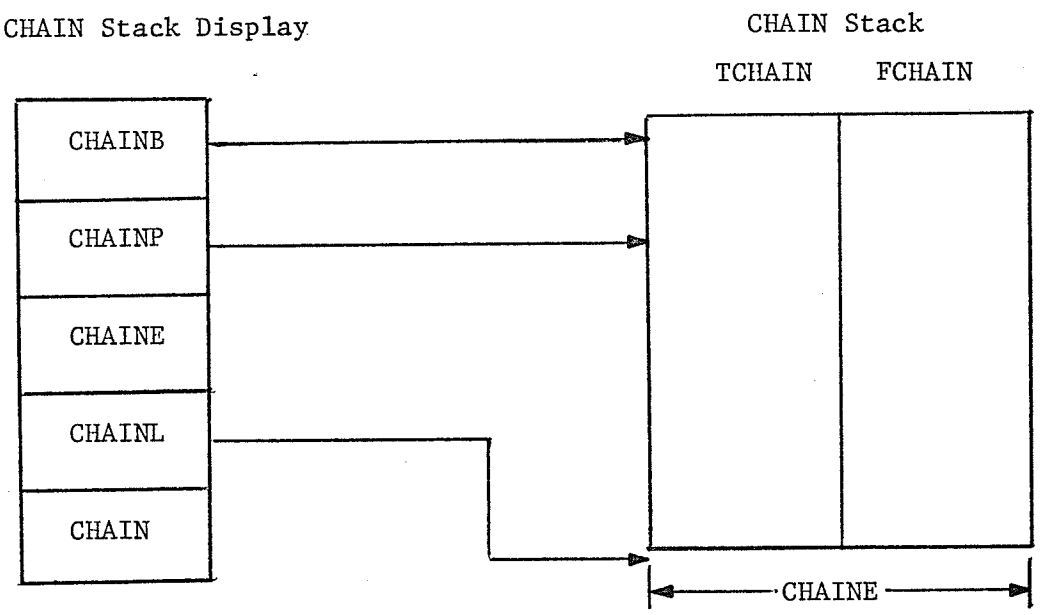
where OFFSET is the value of INSCOUNTER (current displacement in the segment) is constructed. This OFFSET is used at the end of the statement in the address part of a branch instruction as the displacement from register 15.

Forward Branches

Often with forward branches, a whole chain of branch addresses must be filled in rather than the address for just a single branch instruction. This is especially true with complex BOOLEAN expressions and CASE statements.

When the address of the object of these branch chains is discovered the address portions of previously generated branch instructions are filled in. OFFSET gives the location of the address part of the most recent of these branch instructions. A back chain is maintained with the address part of this branch instruction containing the location of the address part of the previous branch instruction in this chain. The chain ends when the address part of a branch instruction contains zero.

CHAIN STACK



CHAINB - base address of CHAIN stack

CHAINP - pointer to first free entry

CHAINE - entry size (4 bytes)

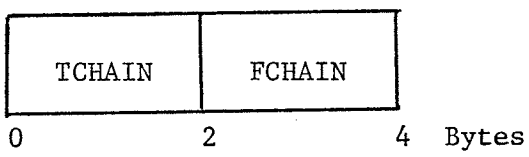
CHAINL - initially the length of the CHAIN stack

- later the address of the end of the CHAIN stack

CHAIN - current backchain information

- when an open parenthesis is encountered in an expression, the value of CHAIN is stored on the chain stack and CHAIN is zeroed. CHAIN is restored from the CHAIN stack when the close parenthesis is reached.

CHAIN is of the form:



The information maintained in CHAIN and the CHAIN stack allows the address part of branch instructions to be resolved. TCHAIN and FCHAIN indicate the locations in the code of the address parts of the most recent branch instructions if the BOOLEAN expression is TRUE or FALSE respectively. TCHAIN therefore indicates the chain of branch instructions, one of which is executed if the BOOLEAN expression is TRUE, while FCHAIN indicates the branch instructions to be taken if the expression is FALSE.

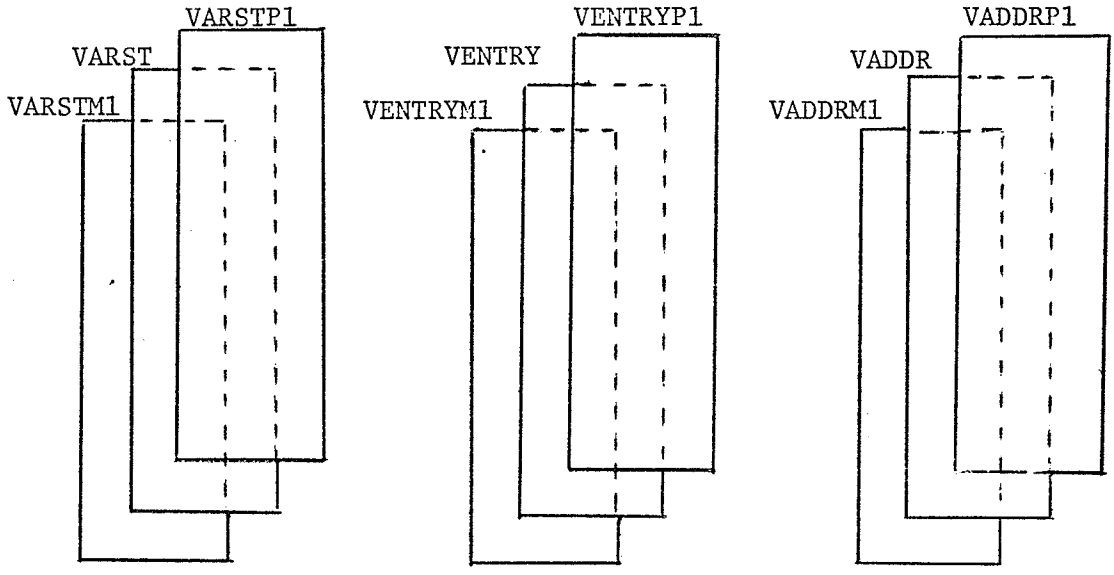
This use of the CHAIN stack allows the TRUE/FALSE decision to be made at the first possible point (evaluating from left to right), without any unnecessary operands being evaluated.

5.7 REGISTER ALLOCATION

The register allocation mechanism and operand stack format are based on those used in the Algol W compiler (Bau68b). This scheme, with some modifications and improvements, seems ideally suited for one-pass code generation. Many of the procedure names used are the same as the corresponding routines in the Algol W compiler. The operation of the operand stack and register allocation mechanism is described in detail noting the major changes to the Algol W scheme.

The primary function of the operand stack and register allocation routines is to control code generation for the evaluation of expressions.

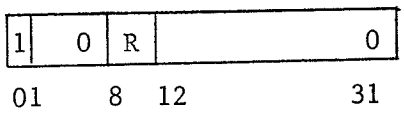
The operand stack is extended to 3 parallel stacks allowing more information to be maintained for compile-time checks.



All entries are full-words. VARSTP is the operand stack pointer. When loaded into a register (say R7) and used as an index, the 3 fields associated with any operand are accessible, e.g. VARST(R7), VENTRY(R7) and VADDR(R7).

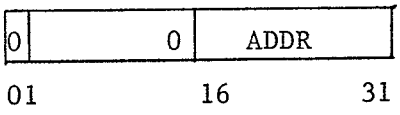
There are two formats for VARST entries:

(1)



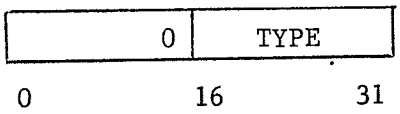
The sign bit is 1 indicating that the operand is in a register. R specifies the register number and may be either a general purpose register or a floating point register.

(2)



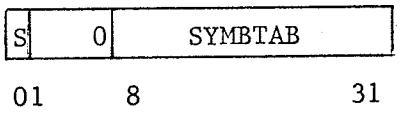
The sign bit is 0 indicating that the operand is not in a register. ADDR is in base-displacement form and is usually entered in the operand stack from the DLOC field of a symbol table entry.

The VENTRY part of the operand stack contains the type of the entry in the corresponding VARST entry. The form is:



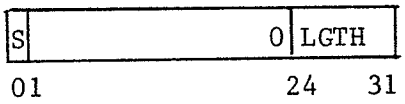
TYPE is one of the internal type codes (CSHRTINT - CRECORD) and is usually entered in the operand stack from the TYPE field of a symbol table entry.

The VADDR part of the operand stack has several forms. The basic format is the following:



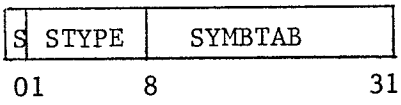
where SYMTAB is the address of the symbol table entry corresponding to VARST. This entry is only used if VENTRY is one of the following: CSET, CSCALAR, CSCELEMENT, CSUBRANGE, CPOINTER, CARRAY or CRECORD. S is the sign bit and is 1 if the VARST entry is an address and is 0 if it is a value.

If the operand is a STRING the following form is used:



LGTH is the length - 1 of the string. This is useful as the length part of a character instruction.

If VENTRY is CSET then a different form again is used:



S and SYMBTAB are as above while STYPE is the base type of the set. (e.g. CINTEGER, CSCALAR). This is used to check that set members are compatible when sets are constructed.

Operands are added to the stack when they are encountered and are removed or modified as required when operations are performed on them.

Normally the top operand on the stack is at the middle (or VARST) position of the three which are accessible at any time. Unary operations

are therefore performed on the VARST entry while binary operations are performed on the VARSTM1 and VARST entries. The operand stack pointer, VARSTP is adjusted up when an operand is added to the stack and down when one is removed. The only time the three positions VARSTM1, VARST and VARSTP1 are required is when compatibility checks are made in FOR statements to see that the initial value and limit are both compatible with the control variable. In any segment, general purpose registers 2 to (11 - BNBR) and all four floating point registers are available as work registers.

The allocation mechanism is more complicated than that used for ALGOLW but allows more flexibility. In particular, the register allocation routines update the operand stack entry which is to use the register. If a flag called POSITION is SET the register is allocated to the VARSTM1 location while if it is RESET the allocation is made to the VARST location. In addition, it is possible to release any register back to the free pool rather than just the last one allocated.

A vector GPR has one word for each general purpose register. This word contains a zero if the register is free and a pointer to the operand stack if it is allocated.

A single general purpose register is requested by a call on GENREG with POSITION and the current value of VARSTP indicating the operand stack location to be assigned the register. GENREG searches the vector GPR from high to low and allocates the first free register. Similarily pairs

of general registers are requested by a call on PRGENREG which searches GPR from low to high. A DIVIDE and a REMAINDER flag indicate which register of a pair is to be entered on the operand stack.

A procedure FORCEPAIR accepts an operand stack entry (register or address) and with the aid of DIVIDE and REMAINDER flags forces the operand into a pair of registers and enters the required register on the operand stack.

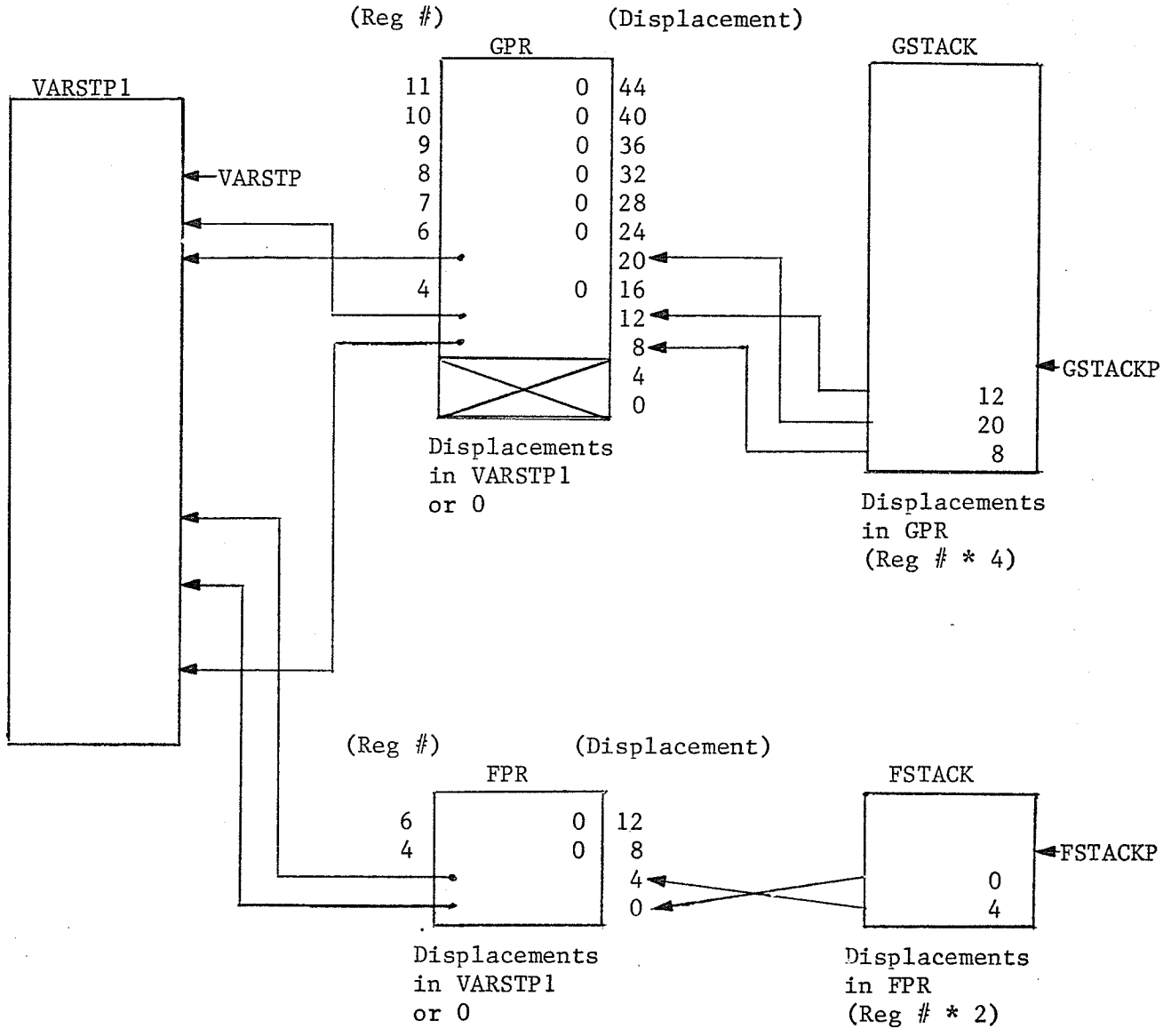
When a general register is allocated an entry is made on a pushdown stack called GSTACK. When a register is requested through a call on GENREG, PRGENREG or FORCEPAIR and none is available, the procedure DUMPGENREG is called. The oldest register (bottom of GSTACK) is freed by storing its contents in the run-time save area and GSTACK is compressed. GSTACK entries are displacements in GPR (i.e. register number * 4) so that when a register is freed the GSTACK value gives the location of the register in GPR.

A vector FPR and pushdown stack FSTACK are used to control floating-point registers. Similarily FPREG and DUMPEPREG correspond to GENREG and DUMPGENREG.

Procedures DALLGREG and DALLFPREG are available to store the contents of all allocated registers. ADJSTACKS decrements the top entry of GSTACK or FSTACK to point to the top entry on the operand stack after a binary operation. ALLOCFO and DGENR2 serve to free floating-point register 0

or general register 2, if occupied, to make them available to hold values returned by calls on functions. RELEASE frees floating-point and general registers which are no longer needed on the operand stack by setting the corresponding entry on FPR or GPR to zero and removing the entry from FSTACK or GSTACK.

REGISTER ALLOCATION MECHANISM



This demonstrates a possible register allocation situation. General registers 2, 3 and 5 as well as floating point registers 0 and 2 are currently allocated to operands.

As the register allocation routines overwrite the operand stack with the register allocated, it is necessary to keep a copy of the previous contents of the operand stack so that the information is available for code generation. This is accomplished by loading the VARST or VARSTM1 entry (address form) into a safe register (one not modified by the allocation routine) before the routine is called.

Procedure ASSEMBLE accepts two VARST type entries (one of which must be a register), type information (CSHRTINT, CINTEGER, CREAL or CLONGREAL) and a code representing the type of instruction to be generated. It will generate RR and RX instructions for at least the following: store, load, compare, add, subtract, multiply and divide.

Compiler routines can either generate machine code directly or they can pass information to ASSEMBLE for code construction. In either case a routine EMIT is called which moves the instruction into the 4K program area.

CHAPTER 6

CODE GENERATED

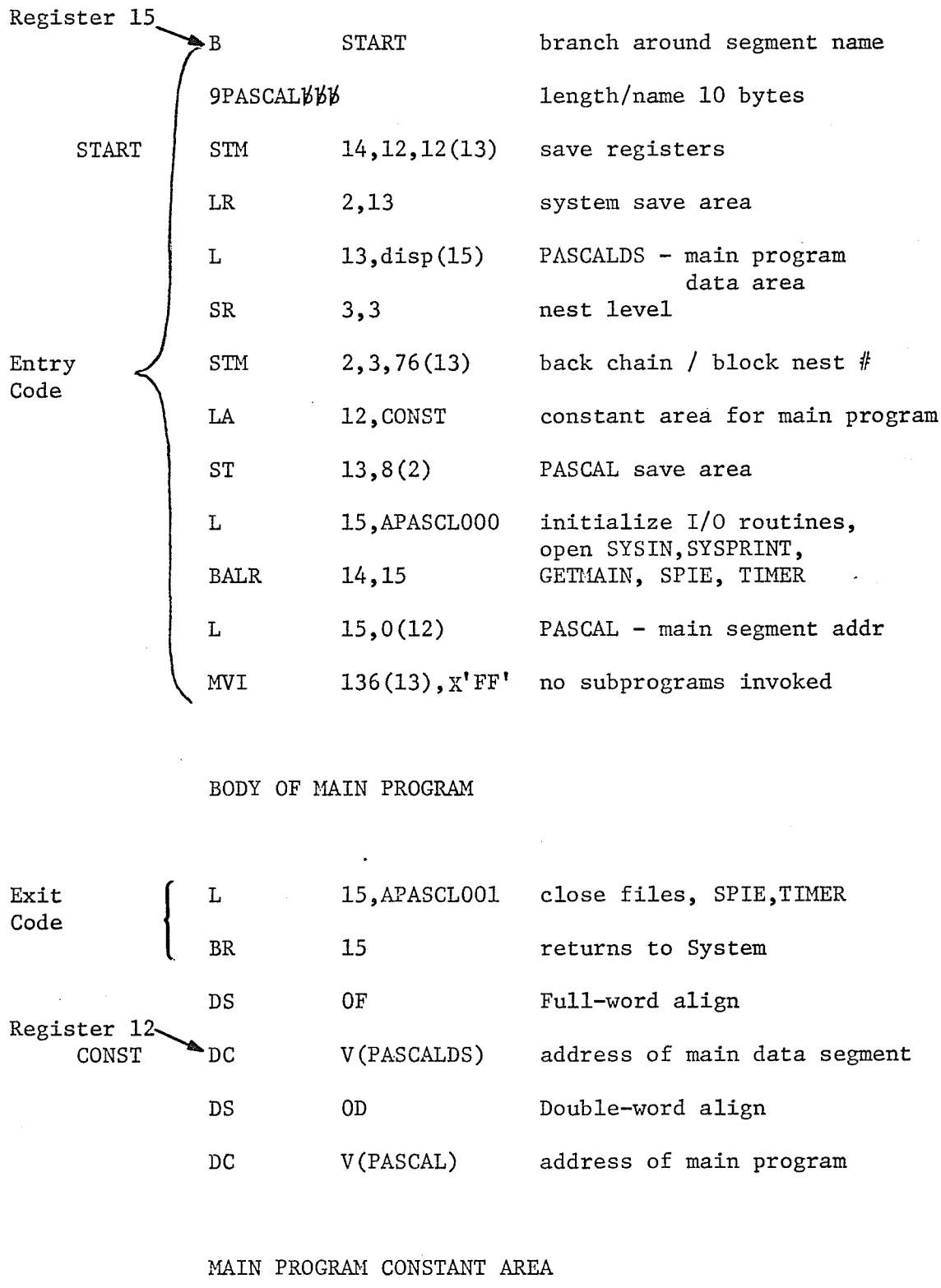
This chapter demonstrates the code which is generated for most constructs of the language. The examples are given in Assembler (IBMa), although the compiler generates machine code.

Initially the entry/exit code is given for both the main program and subprograms. It should be noted that the constant areas follow the program segments directly. A description of PASCALDS (the main program data area) follows, which shows the locations of the save areas, system constants, run-time checking code, etc. The run-time checking code is then listed in detail. It should be noted that the labels (such as ARRAYERR) are used as branch addresses in the checking code of the examples which follow. The save area portion of the subprogram data areas is then shown.

The examples include the code generated for built-in functions, arithmetic conversions and operations, set operations, logical and relational expressions, statements (including IF, WHILE, REPEAT, FOR and CASE), input/output, procedure and function calls, arrays and records (including operation of the WITH statement), pointers and finally assignments.

Where appropriate, the examples are demonstrated with and without run-time checking code.

6.1 MAIN PROGRAM ENTRY / EXIT CODE



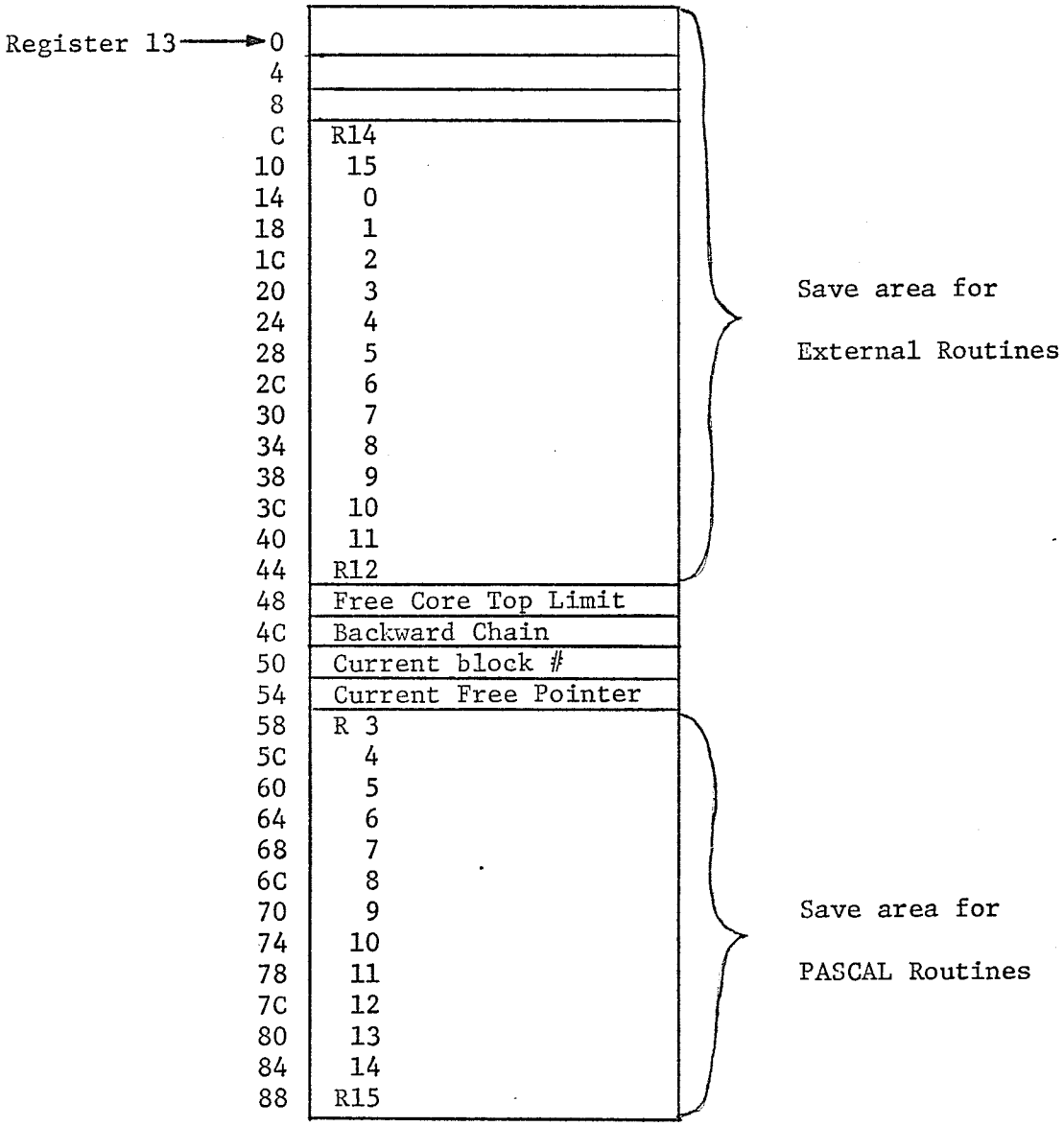
6.2 PROCEDURE / FUNCTION ENTRY / EXIT CODE

n - register for procedure data area = 13 - BNBR

```
Register 15 ->
                B          START          branch around segment name
                9NAMEbbbbbb               length/name  10 bytes
START           STM        3,15,12(2)     Reg 2 - data area of
                                          calling segment
                L          n,8(2)         Procedure data area
                                          current core free pointer
                LR         4,n
                A          4,LENGTH       length of procedure data area
                C          4,72(13)       free core top limit
Entry           BNH        OK
Code
                LA         0,16           available region exceeded
                BAL        1,ERRMSG       error message
OK              LA         3,BNBR         nest level - block number
                STM        2,4,0(n)       back chain/block #/free pointer
                LA         n-1,CONST      reg for constant area
                MVI        60(n),X'FF'    no subprograms invoked

                PARAMETER LIST DECODING

                PROCEDURE / FUNCTION BODY

                L          0,-            function result
                L          2,0(n)         backward chain
Exit            LM         3,14,12(2)     restore registers
Code
                MVI        60(2),X'FF'    control returned
                BR         14             return

                DS         0F             Full-word align
LENGTH          DC         F              length of data area
Register n-1    DS         0D             Double-word align
(12 - BNBR)
CONST           DC         V(NAME)        address of procedure/function

                PROCEDURE / FUNCTION CONSTANT AREA
```

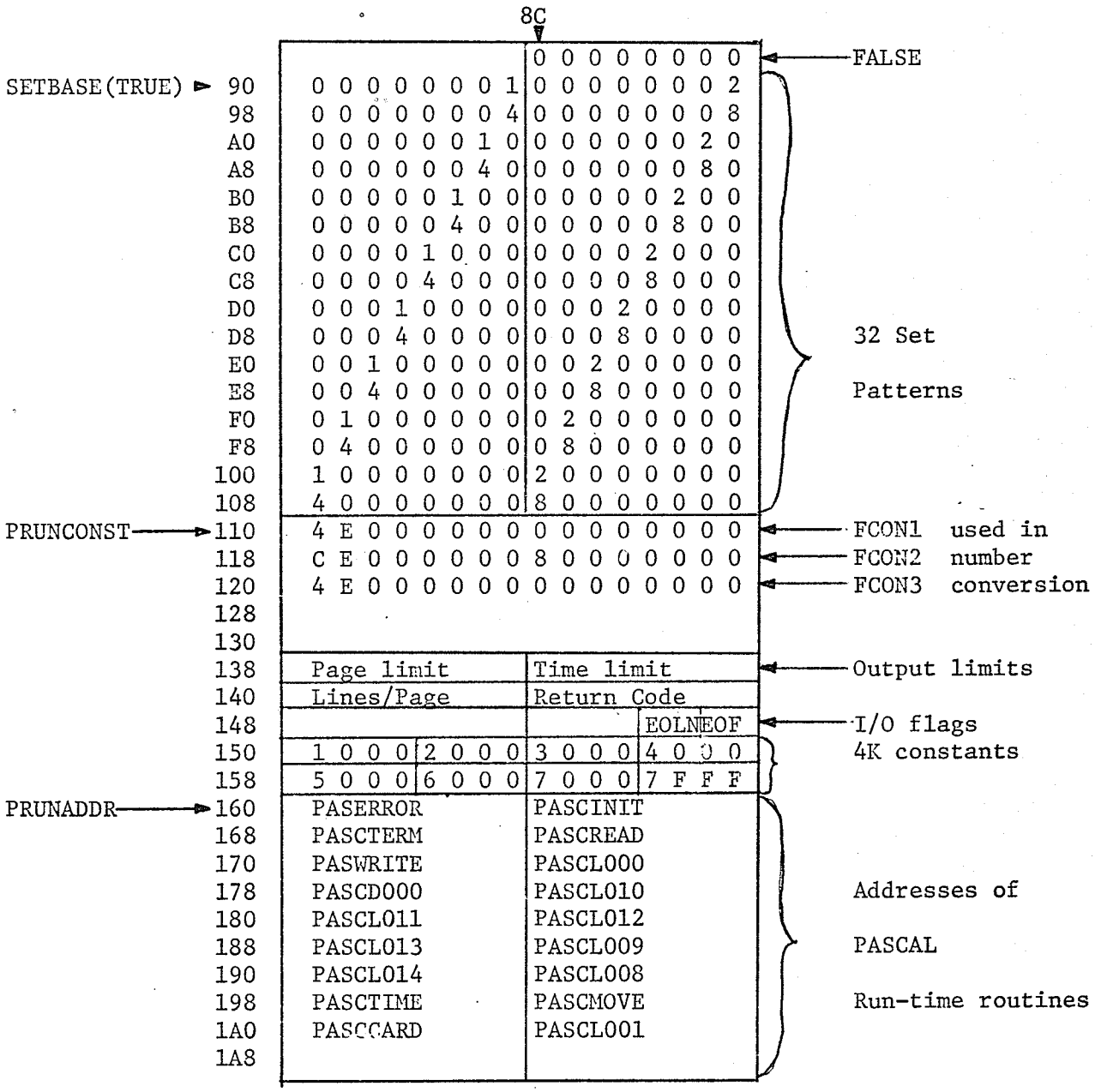
6.3 MAIN PROGRAM DATA AREA (PASCALDS)

- displacements given in hexadecimal

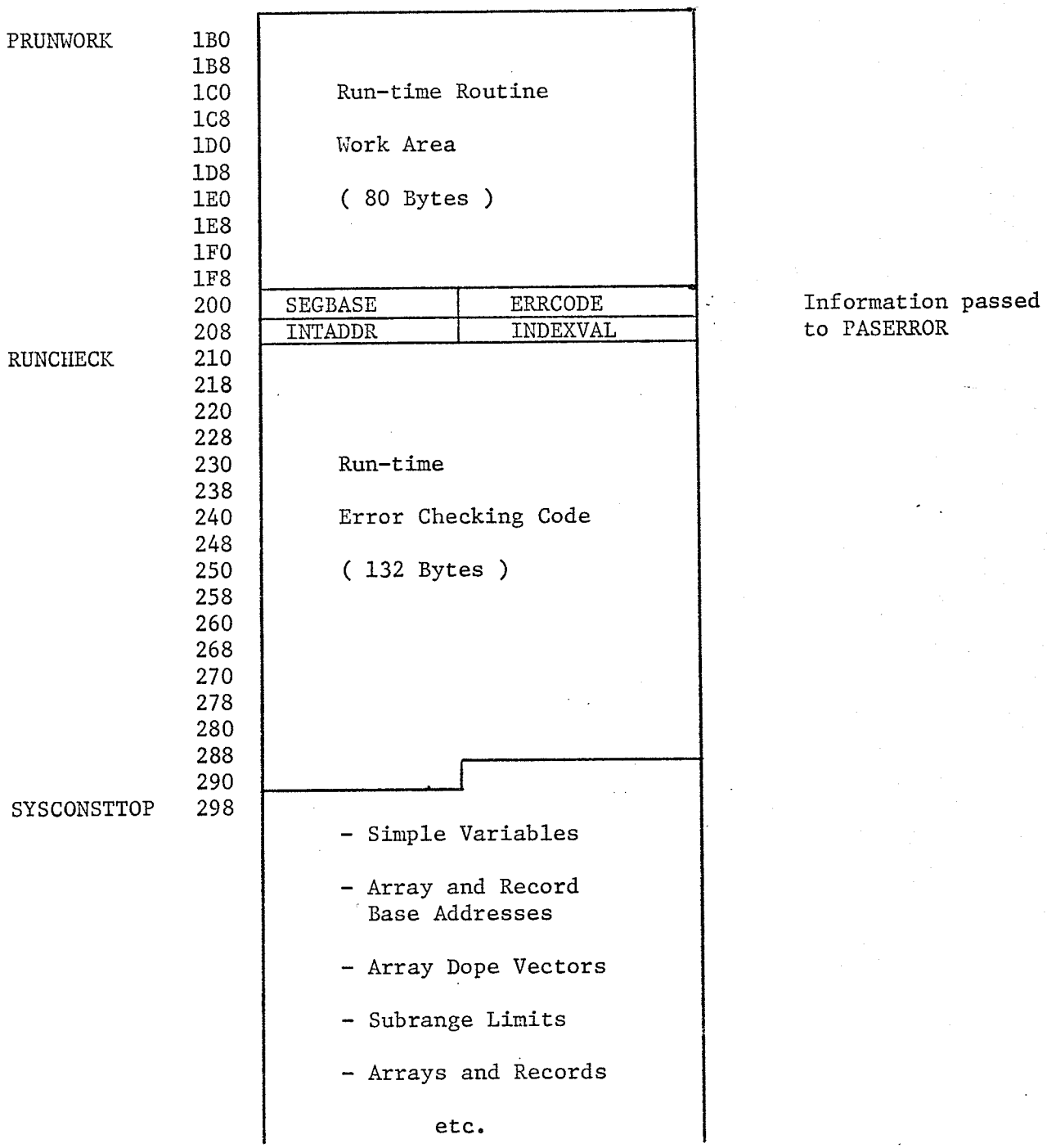


6.3.1 SYSTEM CONSTANTS (in PASCALDS)

- displacements given in hexadecimal



PASCALDS cont'd



6.3.2 RUN-TIME CHECKING CODE IN PASCALDS

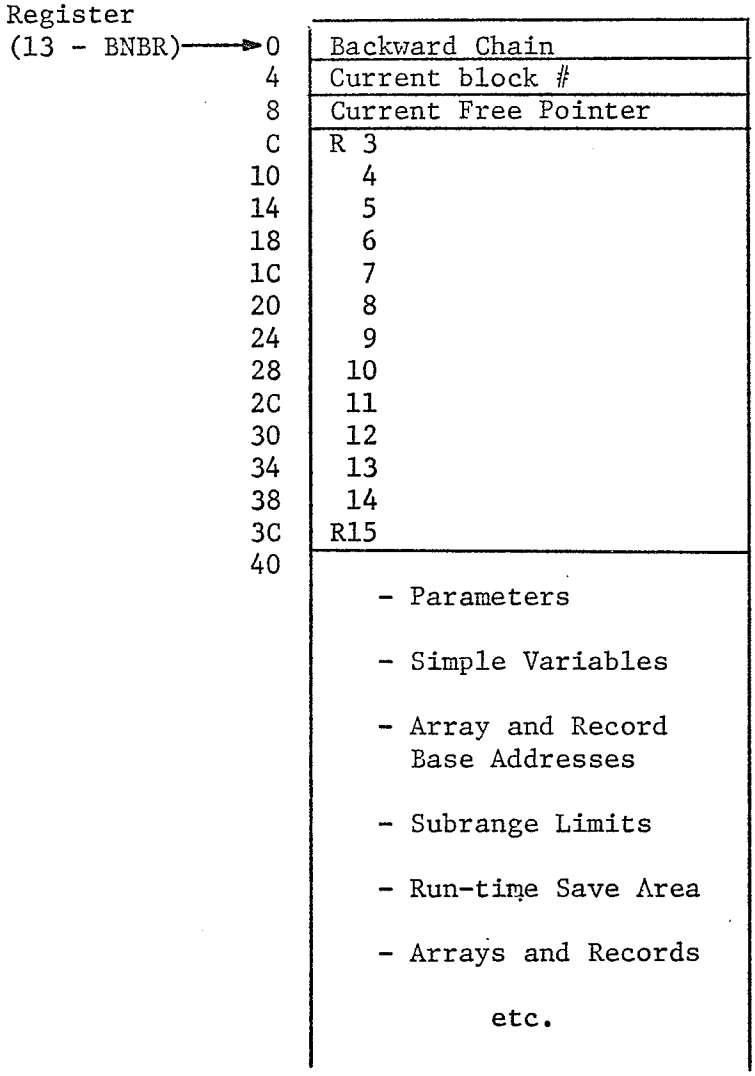
The run-time checking code occupies 132 bytes and starts at RUNCHECK (offset #210) in PASCALDS.

```
ARRAYERR   BNHIR    1              run-time array bounds check(high?)
MAINERR    LR       2,0            (low) - invalid value
           LA       0,35           error message number
ERRMSG     STM      15,2,SEGBASE   store error information
           L        15,APASERROR   run-time error routine
           BR       15             terminate with error
SUBERR     BNHR     1              subrange bounds check (high?)
SUBMERR    LR       2,0            (low) - invalid value
           LA       0,36           error message number
           B        ERRMSG         issue error message
PREDERR    BNLR     1              PRED value undefined?
           LR       2,0            undefined value
           LA       0,38           error message number
           B        ERRMSG         issue error message
SUCCERR    BNHR     1              SUCC value undefined?
           LR       2,0            undefined value
           LA       0,37           error message number
           B        ERRMSG         issue error message
SETERR     BER      1              invalid set member?
           LA       0,34           error message number
           B        ERRMSG         issue error message
```

```
SUBHRD      BNHR    1           subrange value on READ (high?)
SUBLRD      LR      2,0         (low) - invalid value
            LA      0,40        error message number
            B       ERRMSG      issue error message
SUBHPARM    BNHR    1           subrange parameter (high?)
SUBLPARM    LA      0,39        (low) - error message number
                                register 2 contains invalid value
            B       ERRMSG      issue error message
SUBLFUP     BNHR    1           FOR control variable is subrange
                                final value (TO)  (high?)
            LR      2,0         invalid value
            LA      0,42        error message number
            B       ERRMSG      issue error message
SUBIFOR     LR      2,0         FOR control variable is subrange
                                initial value out of range
            LA      0,41        error message number
            B       ERRMSG      issue error message
SUBLFDN     BNLR    1           FOR control variable is subrange
                                final value (DOWNTO)  (low?)
            LR      2,0         invalid value
            LA      0,42        error message number
            B       ERRMSG      issue error message
CASENEG     SRA     2,1         CASE index negative
                                get undefined value
            LA      0,43        error message number
            B       ERRMSG      issue error message
```

6.4 PROCEDURE / FUNCTION DATA AREA

- displacements given in hexadecimal



6.5 BUILT-IN FUNCTIONS

The arithmetic functions are evaluated by passing an argument and function selector to a run-time library routine. These routines are borrowed from the Algol W run-time library. The routine PASCL012 accepts a REAL argument and produces a REAL result, while PASCL013 accepts a LONG REAL argument and produces a LONG REAL result.

The actual parameter used when calling these arithmetic functions may be of any arithmetic type. A SHRTINT or INTEGER argument is converted to a REAL value before PASCL012 is called and produces a REAL result.

Arithmetic Function	Function Selector
SQRT	1
EXP	2
LN	3
LOG	4
SIN	5
COS	6
ARCTAN	7

	Argumement Type	Function Result Type
Arithmetic Function	(SHRTINT)	REAL
	(INTEGER)	
	(REAL)	
	(LONG REAL)	LONG REAL

Example:

```
VAR X : REAL;
BEGIN
   - - -
   X := SQRT(X);
```

The code produced for this statement is:

```
LE      0,X
L       14,APASCL012        address of PASCL012
BALR    1,14
DC      H'1'                function selection code
STE     0,X
```

Register 14 is used as the base register of the arithmetic functions, so that the segment base register (15) does not need to be restored after every call of these routines. The function accesses the selector code using register 1 and returns to the point immediately after the selector code.

BUILT-IN FUNCTIONS HANDLED WITH IN-LINE CODE

The registers used in the examples are those which are used in the main program if no registers are currently allocated.

ABS - Absolute value of argument

	SHRTINT	INTEGER	REAL	LONG REAL
Load register if necessary	LH 11,-	L 11,-	LE 6,-	LD 6,-
Load positive	LPR 11,11	LPR 11,11	LPER 6,6	LPDR 6,6

SQR - Square of argument

	SHRTINT	INTEGER	REAL	LONG REAL
Load register if necessary	LH 3,-	L 3,-	LE 6,-	LD 6,-
Multiply	MR 2,3	MR 2,3	MER 6,6	MDR 6,6

If an integer value is already in a register, it may have to be moved to the odd register of a free even/odd pair.

ODD - TRUE if integer argument is odd

- Result: Argument MOD 2 = 1

	SHRTINT	INTEGER
Load register if necessary	LH 11,-	L 11,-
Mask rightmost bit	N 11,=F'1'	N 11,=F'1'

The result is 1 if the argument is odd and 0 if it is even. This value is directly used as the boolean result, corresponding exactly to the internal representations for TRUE and FALSE respectively.

ORD - Ordinal number

- Argument of type CHAR or user-defined scalar

Example: In the declaration

```
VAR COLOUR : (RED, GREEN, BLUE);
```

COLOUR is a user-defined scalar while BLUE is a scalar constant.

	CHAR	Scalar	Scalar Constant
Code generated if	SR 11,11		
not in register	IC 11,-	L 11,-	LA 11,ordinal number

If the argument is already in a register, no code is generated. In this case, the type information on the operand stack is merely changed to INTEGER.

CHR - Argument must be an integer

- Result is the character whose ordinal number is the argument value

	SHRTINT	INTEGER
Code generated if not in register	LH 11,-	L 11,-

If the argument is already in a register, no additional code is generated. The operand type is merely changed to CHAR.

REAL TO INTEGER CONVERSION

A run-time library routine PASCL009 converts real to integer values. The real value is passed in general register 0 and the required integer value is returned in register 2. A function selector of 1 is used for TRUNC and 2 for ROUND. This is stored in the code after the BALR instruction in the same manner as with the arithmetic functions.

Example:

```
VAR I : INTEGER;  X : REAL;
BEGIN
   - - -
   I := ROUND(X);
```

produces the following code:

```
        L       0,X             load real value
        L       14,APASCL009    address of PASCL009
        BALR    1,14            branch to library routine
        DC      H'2'            function selector
        ST      2,I             store integer result
```

If the statement had been

```
I := TRUNC(X);
```

the only difference would be in the function selector.

If the real argument happens to be in a register, it is stored and then loaded into general register 0.

CARDINALITY OF SETS

The function CARD (PASCCARD) is a run-time library routine which determines the cardinality (the number of members) of a set. It accomplishes this by repeated left shifts, counting the number of times the sign bit is 1. The function accepts the set in general register 0 and returns the cardinality (an integer value) in register 2.

Example:

```
VAR I : INTEGER;  S : SET OF 1..31;
BEGIN
   - - -
   I := CARD(S);
```

produces the following code:

```
        L       0,S                 set
        L       14,APASCCARD        address of PASCCARD
        BALR    1,14
        ST      2,I                 cardinality of set
```

CPU TIMER

The built-in function CPUTIME is used without arguments. A timer is started when execution begins. A call of CPUTIME returns the elapsed time in seconds as a real value in floating point register 0.

Example:

```
VAR X : REAL;
BEGIN
   X := CPUTIME;
```

produces the following code:

```
        L       15,APASCTIME        address of CPUTIME routine
        BALR    14,15               interrogate timer
        L       15,-                restore base register
        STE     0,X                 elapsed time in seconds
```

6.6 ARITHMETIC CONVERSION

Real to integer conversion is handled through the use of functions ROUND and TRUNC. All other arithmetic conversion is handled with in-line code.

The following declarations are assumed in the examples:

```
VAR I : INTEGER;  K : SHRTINT;  X : REAL;  Z : LONG REAL;
```

SHORT INTEGER / INTEGER Conversion

```
I := K;
```

A SHORT INTEGER is changed to an INTEGER by merely loading it into a register:

```
        LH      11,K
        ST      11,I
```

```
K := I;
```

The only time an INTEGER must be changed to SHORT INTEGER is on assignment:

```
        L       11,I
        STH     11,K
```

REAL / LONG REAL Conversion

```
X := Z;
```

A LONG REAL is converted to REAL by just considering the left 32 bits of the LONG REAL operand:

```
        LE      6,Z
        STE     6,X
```

```
Z := X;
```

A REAL value is converted to LONG REAL by loading a double word constant into a floating point register and then loading the REAL value into the same register:

```
            LD      6,FCON3
            LE      6,X
            STD     6,Z
            - - -
FCON3       DC      X'4E00000000000000'
```

If the real value is already in a register as in the statement

```
Z := SQR(X);
```

the code is

```
            LE      6,X
            MER     6,6             square X
            STE     6,FCON3
            LD      6,FCON3         convert to long real
            STD     6,Z
            - - -
FCON3       DC      X'4E00000000000000'
```

The left word of FCON3 does get modified during execution but the right word is always 0.

INTEGER to REAL / LONG REAL Conversion

The INTEGER is converted to double precision floating point so the value can be used as either a REAL or LONG REAL operand.

Four possibilities exist:

```
X := I;  X:= K;  Z := I;  Z := K;
```

The following code demonstrates all cases:

```
I                                                        K
     L      11,I                          LH     11,K
            ST       11,FCON1+4
            XI       FCON1+4,X'80'                    convert integer
            LD       6,FCON1                          to real
            AD       6,FCON2
     STE    6,X                           STD    6,Z
X           - - -                                     Z
     FCON1  DC       X'4E00000000000000'
     FCON2  DC       X'CE00000080000000'
```

6.7 SUCC / PRED FUNCTIONS

The functions SUCC and PRED are handled with in-line code. The valid argument types are integers, user-defined scalars (including scalar constants) and subranges of either integers or user-defined scalars.

It is possible for these functions to produce undefined results in the case of user-defined scalars or subranges. For this reason, run-time checking code is optionally available through the use of the $CHECK+ command.

Without Run-Time Checking Code ($CHECK-)

Integers

	SUCC		PRED	
	SHRTINT	INTEGER	SHRTINT	INTEGER
Load register if necessary	LH 11,-	L 11,-	LH 11,-	L 11,-
	A 11,=F'1'	A 11,=F'1'	BCTR 11,0	BCTR 11,0

User-Defined Scalars

Assume the declaration:

```
VAR TOY = (CAR, BALL, BOAT, BAT);
```

The following demonstrates the code when the argument is a scalar:

SUCC(TOY)	PRED(TOY)
L 11,TOY	L 11,TOY
LA 11,1(11)	BCTR 11,0

The LA instruction can be used for SUCC because a scalar value is always positive.

If the argument is a scalar constant, a compile-time check is made to see if the result is undefined, in which case an error message is given. The function result is produced directly.

SUCC(BAT) and PRED(CAR)

would result in error messages. The statements

TOY := SUCC(BALL); and TOY := BOAT

produce identical code:

```
LA          11,2        ordinal number
ST          11,TOY
```

Likewise,

TOY := PRED(BALL) and TOY := CAR

are the same:

```
LA          11,0        ordinal number
ST          11,TOY
```

Subranges

If run-time checking is not performed, then the code generated for subranges of integers and user-defined scalars is identical to that produced for integers and scalars discussed above.

With Run-Time Checking Code ($CHECK+)

Integers

The code is the same as with $CHECK-. All results are considered valid. A check is not made to see if MAXINT is exceeded.

User-Defined Scalars

The function results are computed normally, but then a check is made to see if PRED produces a negative value (the ordinal number of the first scalar constant is 0), or if SUCC produces a result higher than the ordinal number of the last scalar constant.

Assuming the same declarations as above,

```
TOY := SUCC(TOY);
```

generates:

```
L       11,TOY
LA      11,1(11)
LR      0,11
LA      1,highest ordinal   } $CHECK+
CR      0,1
BAL     1,SUCCERR
```

An additional 4 instructions are generated in-line and a total of 5 extra instructions are executed to check if the result is valid.

```
TOY := PRED(TOY);
```

generates:

```
L       11,TOY
BCTR    11,0
LTR     0,11        } $CHECK+
BAL     1,PREDERR
```

An additional 2 instructions are generated in-line and 3 instructions are executed to check the resultant value.

Subranges

The function results are computed and then are compared to the lower bound of the subrange for PRED or to the upper bound for SUCC. Assume the declarations:

```
TYPE COLOUR = (RED, GREEN, BLUE, YELLOW, BLACK);
VAR M : GREEN..YELLOW;
    I : 1..10;
```

The following code is generated for SUCC:

```
SUCC(M)                          SUCC(I)
L 11,M                           L 11,I
LA 11,1(11)                      A 11,=F'1'
                  LR 0,11      )
                  C 0,upper    } $CHECK+
                  BAL 1,SUCCERR)
```

The following code is generated for PRED:

```
PRED(M)                          SUCC(I)
L 11,M                           L 11,I
                  BCTR 11,0
                  LR 0,11      )
                  C 0,lower    } $CHECK+
                  BAL 1,PREDERR)
```

These four cases require 3 extra instructions in-line, and the execution of 4 extra instructions to perform the required checks.

6.8 ARITHMETIC OPERATIONS

Before code is generated for any binary arithmetic operation, a routine is called which checks operand type compatibility. This routine performs any necessary type conversions and sets the condition code to indicate whether or not the arithmetic operation can proceed.

The four standard numeric types, SHRTINT, INTEGER, REAL and LONG REAL are handled. The binary operations are divided into four categories according to the following table:

Category	Operator
1	*, +, -
2	/
3	DIV, MOD
4	:=

A conversion table, when indexed with the numeric types of the left and right operands, along with the category code, returns a conversion code which indicates the action to be taken.

CONVERSION TABLE

Right Operand

Left Operand	Category Codes	SHRTINT 1	SHRTINT 2	SHRTINT 3	SHRTINT 4	INTEGER 1	INTEGER 2	INTEGER 3	INTEGER 4	REAL 1	REAL 2	REAL 3	REAL 4	LONG REAL 1	LONG REAL 2	LONG REAL 3	LONG REAL 4
	SHRTINT	0	21	20	0	0	21	1	0	2	2	10	10	3	3	10	10
	INTEGER	0	21	11	0	0	21	0	0	4	4	10	10	5	5	10	10
	REAL	12	12	10	12	14	14	10	14	0	0	10	0	7	7	10	19
	LONG REAL	13	13	10	13	15	15	10	15	17	17	10	17	0	0	10	0

Conversion Codes

CONVERSION CODES

Operand	Code	Action			
Neither	0	No conversion necessary			
Left	1	SHRTINT	to	INTEGER	
"	2	"	to	REAL	
"	3	"	to	LONG REAL	
"	4	INTEGER	to	REAL	
"	5	"	to	LONG REAL	
"	6	REAL	to	INTEGER	(unused)
"	7	"	to	LONG REAL	
"	8	LONG REAL	to	INTEGER	(unused)
Left	9	"	to	REAL	(unused)
Neither	10	Illegal operation (ERROR)			
Right	11	SHRTINT	to	INTEGER	
"	12	"	to	REAL	
"	13	"	to	LONG REAL	
"	14	INTEGER	to	REAL	
"	15	"	to	LONG REAL	
"	16	REAL	to	INTEGER	(unused)
"	17	"	to	LONG REAL	
"	18	LONG REAL	to	INTEGER	(unused)
Right	19	"	to	REAL	(unused)
Both	20	Convert from SHRTINT to INTEGER			
Both	21	Convert from SHRTINT or INTEGER to REAL			

In the examples R2 indicates general register 2, while F2 indicates floating-point register 2 containing a single precision value, and F23 indicates floating-point register 2 containing a double precision value. If an operand is in memory (i.e., not in a register), it is represented by its name.

ADDITION

Addition is commutative. If the left operand is in a register, a RR or RX add instruction is generated depending on whether or not the right operand is in a register.

If the left operand is not in a register, but the right operand is in a register, the suitable RX add instruction is generated.

If neither operand is in a register, the left operand is loaded into a register and the suitable RX add instruction is generated.

A + B

Both Operands in Registers

	INTEGER	REAL	LONG REAL
Location of: A	R11	F6	F67
B	R10	F4	F45
Code generated:	AR 11,10	AER 6,4	ADR 6,4

Left Operand in Register

	SHRTINT	INTEGER	REAL	LONG REAL
Location of: A	R11	R11	F6	F67
B	B	B	B	B
Code generated:	AH 11,B	A 11,B	AE 6,B	AD 6,B

Right Operand in Register

	SHRTINT	INTEGER	REAL	LONG REAL
Location of: A	A	A	A	A
B	R11	R11	F6	F67
Code generated:	AH 11,A	A 11,A	AE 6,A	AD 6,A

Neither Operand in Registers

	SHRTINT	INTEGER	REAL	LONG REAL
Location of: A	A	A	A	A
B	B	B	B	B
Code generated:	LH 11,A	L 11,A	LE 6,A	LD 6,A
	AH 11,B	A 11,B	AE 6,B	AD 6,B

SUBTRACTION

Subtraction is not commutative. For this reason it is necessary for the left operand to be in a register before the operation can proceed.

A - B

Both Operands in Registers

	INTEGER	REAL	LONG REAL
Location of: A	R11	F6	F67
B	R10	F4	F45
Code generated:	SR 11,10	SER 6,4	SDR 6,4

Left Operand in Register

	SHRTINT	INTEGER	REAL	LONG REAL
Location of: A	R11	R11	F6	F67
B	B	B	B	B
Code generated:	SH 11,B	S 11,B	SE 6,B	SD 6,B

Right Operand in Register

	SHRTINT	INTEGER	REAL	LONG REAL
Location of: A	A	A	A	A
B	R11	R11	F6	F67
Code generated:	LH 10,A	L 10,A	LE 4,A	LD 4,A
	SR 10,11	SR 10,11	SER 4,6	SDR 4,6

Neither Operand in Registers

	SHRTINT	INTEGER	REAL	LONG REAL
Location of: A	A	A	A	A
B	B	B	B	B
Code generated:	LH 11,A	L 11,A	LE 6,A	LD 6,A
	SH 11,B	S 11,B	SE 6,B	SD 6,B

MULTIPLICATION

Multiplication is commutative, and therefore can proceed if either operand is in a register. Integer multiplication requires an even/odd pair of registers and is therefore a special case. The MH instruction is used if either or both of the operands is of type SHRTINT.

A * B

Both Operands in Registers

	REAL	LONG REAL
Location of: A	F6	F67
B	F4	F45
Code generated:	MER 6,4	MDR 6,4

Left Operand in Register

	SHRTINT	REAL	LONG REAL
Location of: A	R11	F6	F67
B	B	B	B
Code generated:	MH 11,B	ME 6,B	MD 6,B

Right Operand in Register

	SHRTINT	REAL	LONG REAL
Location of: A	A	A	A
B	R11	F6	F67
Code generated:	MH 11,A	ME 6,A	MD 6,A

Neither Operand in Registers

	SHRTINT	REAL	LONG REAL
Location of: A	A	A	A
B	B	B	B
Code generated:	LH 11,A	LE 6,A	LD 6,A
	MH 11,B	ME 6,B	MD 6,B

INTEGER MULTIPLICATION

The multiplicand must be forced into the odd register of an even/odd pair. If the left operand is in a register, or failing that, if the right operand is in a register, an attempt is made to use the even/odd pair of which this register is a part. If the necessary register is occupied, a pair of registers is allocated and the value is moved into the odd register of this pair.

Operand in Odd Register and Even Register Free

Location of: A	R11	A	R11
B	B	R11	R9
Code generated:	M 10,B	M 10,A	MR 10,9

Operand in Even Register and Odd Register Free

Location of: A	R10	A	R10
B	B	R10	R9
Code generated:	SRDA 10,32	SRDA 10,32	SRDA 10,32
	M 10,B	M 10,A	MR 10,9

Operand in Register but Pair Not Free

Location of:	A	R10	A	R10
	B	B	R11	R11
Code generated:		LR 3,10	LR 3,11	LR 3,10
		M 2,B	M 2,A	MR 2,11

Neither Operand in Registers

Location of:	A	A
	B	B
Code generated:		L 3,A
		M 2,B

Integer multiplication is always followed by

```
SLDA   R,32
```

where R is the even register of the product even/odd pair. This is done to detect an overflow on integer multiplication. An overflow is not detected if the MH instruction is used because of a SHRTINT operand.

DIVISION

There are three division operators. The "/" always produces a real result. If one or more of the operands is of type integer, it is converted to real before the operation takes place. For this reason the code sequences are shown for real operands.

A / B

REAL Operands

Location of: A	F6	A	F6	A
B	B	F6	F4	B
Code generated:	DE 6,B	LE 4,A	DER 6,4	LE 6,A
		DER 4,6		DE 6,B

LONG REAL Operands

Location of: A	F67	A	F67	A
B	B	F67	F45	B
Code generated:	DD 6,B	LD 4,A	DDR 6,4	LD 6,A
		DDR 4,6		DD 6,B

INTEGER DIVISION

The operators DIV and MOD are used for division of integer operands. The result of DIV is the quotient, while MOD gives the remainder. The code sequences are the same in both cases. The result of DIV is in the odd register, while the result of MOD is in the even register of the even/odd pair.

The code sequences to prepare for integer division are similar to those for integer multiplication, with the added requirement that the dividend must be a 64 bit number in the even/odd pair. As division is not commutative, the left operand must be loaded into the even/odd pair.

$$A \left\{ \begin{matrix} \text{DIV} \\ \text{MOD} \end{matrix} \right\} B$$

Left Operand in Odd Register and Even Register Free

Location of: A	R11	R11
B	B	R9
Code generated:	LR 10,11	LR 10,11
	SRA 10,31	SRA 10,31
	D 10,B	DR 10,9

Left Operand in Even Register and Odd Register Free

Location of: A	R10	R10
B	B	R9
Code generated:	SRDA 10,32	SRDA 10,32
	D 10,B	DR 10,9

Left Operand in Register but Pair Not Free

Location of: A	R10	R11
B	B	R10
Code generated:	LR 2,10	LR 2,11
	SRDA 2,32	SRDA 2,32
	D 2,B	DR 2,10

Left Operand Not in Register

Location of: A	A	A
B	B	R11
Code generated:	L 2,A	L 2,A
	SRDA 2,32	SRDA 2,32
	D 2,B	DR 2,11

UNARY MINUS

If the operand is a constant (SHRTINT, INTEGER, REAL or LONG REAL), the constant value is complemented and a new constant is created at compile time.

The statements:

```
A := 10;
B := -10;
```

therefore have the same code sequence at run time.

The complementing is performed at run time if the operand is a variable or expression.

- A

	SHRTINT	INTEGER	REAL	LONG REAL
Load register if necessary	LH 11,A	L 11,A	LE 6,A	LD 6,A
	LCR 11,11	LCR 11,11	LCER 6,6	LCDR 6,6

6.9 SETS

Each set member is represented by one bit. The bits are allocated from right to left with the right-most bit representing the integer 0, or a scalar constant with ordinal number 0. The data area for the main program (PASCALDS) contains the 32 possible set member representations starting at location SETBASE. These are used directly when set members are constants, or are accessed by computing the address at run time when a set member is a variable.

Assume the following declarations:

```
TYPE COLOUR = (RED, GREEN, BLUE, PURPLE, PINK, BLACK);
VAR C : COLOUR;
    HUE1, HUE2 : SET OF COLOUR;
```

The code generated for

```
(. RED, BLUE, BLACK .)
```

is:

```
        L       11,SETBASE          (.RED.)
        O       11,SETBASE+2*4      (.RED,BLUE.)
        O       11,SETBASE+5*4      (.RED,BLUE,BLACK.)
```

The code generated for

```
(. RED, C .)
```

is:

```
        L       11,SETBASE          (.RED.)
        L       10,C
        SLL     10,2
        L       10,SETBASE(10)      (.C.)
        OR      11,10               (.RED,C.)
```

The scalar value is multiplied by 4 (shifted left 2) and then this value is used as an index (displacement from SETBASE) to select the required set mask.

The representation M..N indicates all of the members from M to N inclusive. If M is greater than N, the null set is produced. If both M and N are constants, the set is determined at compile time and is stored in the constant area of the current segment.

The code generated for

```
(. RED .. PURPLE .)
```

is:

```
        L       11,set constant     (.RED,GREEN,BLUE,PURPLE.)
```

If either, or both, of M and N are variables, the set is constructed at run time.

The code generated for

```
(. RED .. C .)
```

is:

```
        LA      11,0                RED
        L       10,C
        SR      9,9                 (..)
        SLL     11,2
        SLL     10,2
LOOP    CR      11,10
        BH      LEND
        O       9,SETBASE(11)       set masks
        LA      11,4(11)
        B       LOOP
LEND
```

Register 9 contains the required set.

SET OPERATIONS

Before code is generated for any set operation, a routine is called to ensure that the operands are both sets, and of compatible types. Singleton sets are constructed if necessary.

UNION

Set union is commutative. The code sequences are similar to those for integer addition.

S1 + S2

Location of: S1	R11	S1	R11	S1
S2	S2	R11	R10	S2
Code generated:				L 11,S1
	O 11,S2	O 11,S1	OR 11,10	O 11,S2

INTERSECTION

Set intersection is commutative. The code sequences are similar to those for set union.

S1 * S2

Location of: S1	R11	S1	R11	S1
S2	S2	R11	R10	S2
Code generated:				L 11,S1
	N 11,S2	N 11,S1	NR 11,10	N 11,S2

SET DIFFERENCE

Set difference is not commutative. If the left operand is in a register, it must not be destroyed until the operation is complete.

S1 - S2

Location of: S1	R11	S1	R11	S1
S2	S2	R11	R10	S2
Code generated:	L 10,S2			L 11,S2
	NR 10,11	N 11,S1	NR 10,11	N 11,S1
	XR 10,11	X 11,S1	XR 10,11	X 11,S1

SET MEMBERSHIP

A test is made to see if the first operand is a member of the second. The result is a boolean value; TRUE if it is present, and FALSE if it is not. The boolean value is not constructed. A branch instruction is generated which causes a branch to be taken if the first operand is not present in the set (i.e., result is FALSE).

X IN S

Location of: X	R11	X	R11	X
S	S	R11	R10	S
Code generated:				L 11,X
	N 11,S	N 11,X	NR 11,10	N 11,S
	BZ false	BZ false	BZ false	BZ false

SET INCLUSION

A test is made to see if one set is entirely contained within the other. Similar to set membership, the result is a boolean value. This boolean value is not constructed but is reflected in a branch instruction. The branch is taken if one set is not included in the other (i.e., result is FALSE).

S1 <= S2

Location of: S1	R11	S1	R11	S1
S2	S2	R11	R10	S2
Code generated:	L 1,S2			L 1,S2
	NR 1,11	N 11,S1	NR 10,11	N 1,S1
	CR 1,11	C 11,S1	CR 10,11	C 1,S1
	BNE false	BNE false	BNE false	BNE false

S1 >= S2

Location of: S1	R11	S1	R11	S1
	S2	R11	R10	S2
Code generated:		L 1,S1		L 1,S1
	N 11,S2	NR 1,11	NR 11,10	N 1,S2
	C 11,S2	CR 1,11	CR 11,10	C 1,S2
	BNE false	BNE false	BNE false	BNE false

SET ASSIGNMENT ($CHECK+)

A mask containing a bit to represent every valid set member is created at compile time for every set which is declared. This mask is used at run time to detect invalid set members on assignment.

The assignment

S1 := set

generates:

```
L       11,-                      if necessary
LR      0,11        }
N       0,mask      }
                    }   $CHECK+
CR      11,0        }
BAL     1,SETERR    }
ST      11,S1
```

An additional 4 instructions are generated in-line and a total of 5 extra instructions are executed to ensure that the set contains only members which are valid.

6.10 LOGICAL (BOOLEAN) EXPRESSIONS

Logical expressions are evaluated from left to right. The TRUE/FALSE decision is made as soon as possible and therefore no unnecessary operands are evaluated.

A BOOLEAN value is not normally constructed, but is reflected in the branch instructions which are generated. If the BOOLEAN result is needed, in a case such as assignment, it is constructed with additional code.

The statements IF, WHILE and REPEAT are designed in such a way that a branch is taken if the expression is FALSE. The code for the logical expressions is therefore generated so that execution continues with the next statement if the expression is TRUE but takes a branch if the expression is FALSE. The operator NOT (¬) does not cause any additional code to be generated. Its effect is reflected in the conditions in the branch instructions.

BOOLEAN values are allocated a full-word of memory. FALSE is represented by #00000000, while TRUE is represented by #00000001. These correspond to the ordinal numbers of FALSE and TRUE in the predefined scalar:

```
TYPE BOOLEAN = ( FALSE, TRUE );
```

```
VAR A, B, C : BOOLEAN;
- - -
   A OR B AND C
   A | B & C
```

```
                    L       0,A
                    LTR     0,0
                    BNZ     TRUE
                    L       0,B
                    LTR     0,0
                    BZ      FALSE
                    L       0,C
                    LTR     0,0
                    BZ      FALSE
            TRUE    - - -
```

The T and F chains are filled in when the objects of the branches are resolved.

If a BOOLEAN result is required as in:

```
A := A OR B AND C;
```

the following additional code is generated:

```
            TRUE    LA      0,1
                    B       STORE
            FALSE   SR      0,0
            STORE   ST      0,A
```

```
                    VAR A, B, C, D : BOOLEAN;

                    - - -
```

A AND B AND C OR NOT D

A & B & C |¬D

```
          L       0,A
          LTR     0,0
          BZ      F1
          L       0,B
          LTR     0,0
          BZ      F1
          L       0,C
          LTR     0,0
          BNZ     TRUE
F1        L       0,D
          LTR     0,0
          BNZ     FALSE
TRUE
          - - -
FALSE
```

NOT (A AND B AND C OR NOT D)

¬(A & B & C |¬D)

```
          L       0,A
          LTR     0,0
          BZ      F1
          L       0,B
          LTR     0,0
          BZ      F1
          L       0,C
          LTR     0,0
          BNZ     FALSE
F1        L       0,D
          LTR     0,0
          BZ      FALSE
TRUE
          - - -
FALSE
```

```
VAR A, B, C, D : BOOLEAN;

- - -

    NOT ( A AND B AND NOT ( NOT C OR D ) )

    ¬( A & B & ¬(¬C |D ) )

                              L       0,A
                              LTR     0,0
                              BZ      TRUE
                              L       0,B
                              LTR     0,0
                              BZ      TRUE
                              L       0,C
                              LTR     0,0
                              BZ      TRUE
                              L       0,D
                              LTR     0,0
                              BZ      FALSE
                    TRUE
                              - - -
                    FALSE
```

If A is FALSE, it is the only operand evaluated. If B is FALSE, only A and B are evaluated. All four operands are evaluated only when A, B and C are all TRUE.

6.11 RELATIONAL EXPRESSIONS

The result of a relational operation is a logical value. Similar to logical expressions, the BOOLEAN value is generated only if necessary. The result is normally reflected in the condition of a branch instruction.

Relational operations are normally not commutative. In the case of SHORT INTEGER, INTEGER, REAL, LONG REAL, BOOLEAN, scalar and subrange operands, the comparison is switched if the second operand is in a register and the first is not. In this case, the condition in the branch instruction is modified to reflect the new ordering in the comparison. The operator IN falls into the category of a relational operator, but is discussed with SETs.

Arithmetic Operands

A sample of the code sequences is shown for INTEGER operands.

A <= B

Location of: A	R11	A	R11	A
B	B	R11	R10	B
Code generated:				L 11,A
	C 11,B	C 11,A	CR 11,10	C 11,B
	BH FALSE	BL FALSE	BH FALSE	BH FALSE

If the BOOLEAN result is required, these sequences are followed by:

```
TRUE    LA      0,1
        B       NEXT
FALSE   SR      0,0
NEXT
```

The code sequences for operands of type BOOLEAN, user-defined scalars or pointers are similar to those for INTEGERs, although pointers may only be compared for equality. The comparisons for operands of type SHRTINT, REAL and LONG REAL are similar if the following conversions are made:

INTEGER	SHRTINT	REAL	LONG REAL
L	LH	LE	LD
C	CH	CE	CD
CR	CR	CER	CDR

CHAR and STRING Operands

Comparisons are always made to the length of the shorter operand. All character strings are of fixed lengths which are known at compile time. The maximum size allowed is 256 characters, allowing all comparisons to be made with a single CLC instruction.

```
VAR S : STRING(10);  T : STRING(5);
- - -
    S = T

                        CLC     S(5),T
                        BNE     FALSE
```

```
VAR X, Y : REAL;
    A, B : BOOLEAN;
    S, T : STRING(5);
    COLOUR : (RED, GREEN, BLUE, YELLOW);
    - - -
    ( X < Y ) AND ( A < B ) OR ( S > T ) AND ( COLOUR >= BLUE )
```

```
                    LE      6,X
                    CE      6,Y
                    BNL     F1
                    L       0,A
                    C       0,B
                    BL      TRUE
            F1      CLC     S(5),T
                    BNH     FALSE
                    LA      11,2            BLUE
                    C       11,COLOUR
                    BH      FALSE
            TRUE    - - -
                    - - -
            FALSE
```

6.12 STATEMENTS

6.12.1 IF Statement

The IF statement has two forms:

1/ IF expression THEN statement

and

2/ IF expression THEN statement1 ELSE statement2

```
VAR A : BOOLEAN;
- - -
1/ IF A THEN statement;

                        L       0,A
                        LTR     0,0
                        BZ      FALSE
                TRUE    statement
                FALSE
```

```
2/ IF A THEN statement1 ELSE statement2;

                        L       0,A
                        LTR     0,0
                        BZ      FALSE
                TRUE    statement1
                        B       NEXT
                FALSE   statement2
                NEXT
```

6.12.2 WHILE Statement

The WHILE statement causes a statement to be repeatedly executed as long as a condition holds TRUE.

```
VAR A : BOOLEAN;
- - -
   WHILE A DO statement;

              LOOP      L       0,A
                        LTR     0,0
                        BZ      FALSE
              TRUE      statement
                        B       LOOP
              FALSE
```

If the expression is FALSE on entry to the WHILE, the statement in the range of the WHILE is not executed at all.

6.12.3 REPEAT Statement

The REPEAT statement causes a sequence of statements to be repeatedly executed until a condition becomes TRUE.

```
VAR A : BOOLEAN;
- - -
   REPEAT
      statement1;
      statement2
   UNTIL A;
```

```
                FALSE     statement1
                          statement2
                          L       0,A
                          LTR     0,0
                          BZ      FALSE
                TRUE
```

The statements in the range of the REPEAT are always executed at least once because the test condition is at the end of the statement.

If the expression in the IF, WHILE and REPEAT statements is a logical expression rather than a simple variable, the BOOLEAN result is not constructed. Its value is reflected in the branches which are taken. All three statements are designed in such a way that the branch is taken if the expression is FALSE and execution continues with the next statement if the expression is TRUE.

6.12.4 FOR Statement

The FOR statement has two forms:

1/ FOR control variable := initial value TO final value DO statement

and

2/ FOR control variable := initial value DOWNTO final value DO statement

The valid types for the control variable, initial value and final value are SHRTINT, INTEGER, BOOLEAN, user-defined scalar or a subrange of any of these.

If the control variable is declared to be a subrange of integers, run-time checking code is produced if $CHECK+ is specified. This code compares the initial and final values with the range limits of the subrange on entry to the FOR statement. Subranges of user-defined scalars are checked at compile time.

1/ VAR I, J, K : INTEGER;

\- - -

FOR I := J TO K DO statement;

```
                    L       0,K
                    ST      0,LIMIT         in run-time save area
                    L       0,J
                    B       COMPARE
LOOP                L       0,I
                    A       0,=F'1'
COMPARE             C       0,LIMIT
                    BH      NEXT
                    ST      0,I
                    statement
                    B       LOOP
NEXT
```

2/ TYPE COLOUR = (RED, GREEN, BLUE, YELLOW, BROWN, BLACK);
VAR A, B, C : COLOUR;
- - -
FOR A := B DOWNTO C DO statement;

```
                    L       0,C
                    ST      0,LIMIT     in run-time save area
                    L       0,B
                    B       COMPARE
LOOP                L       0,A
                    BCTR    0,0
COMPARE             C       0,LIMIT
                    BL      NEXT
                    ST      0,A
                    statement
                    B       LOOP
NEXT
```

3/
```
VAR I, J : INTEGER;
    DAY : 1..31;
- - -
   FOR DAY := I TO J DO statement;
```

Checking Code

```
                          L       0,J
$CHECK+                   C       0,=F'31'
$CHECK+                   BAL     1,SUBLFUP
                          ST      0,LIMIT     in run-time save area
                          L       0,I
$CHECK+                   C       0,=F'1'
$CHECK+                   BL      SUBIFOR
                          B       COMPARE
                LOOP      L       0,DAY
                          A       0,=F'1'
                COMPARE   C       0,LIMIT
                          BH      NEXT
                          ST      0,DAY
                          statement
                          B       LOOP
                NEXT
```

4/
```
VAR I, J : SHRTINT;
    DAY : 1S..31S;
- - -
   FOR DAY := I DOWNTO J DO statement;
```

Checking Code

```
                          LH      0,J
$CHECK+                   CH      0,=H'1'
$CHECK+                   BAL     1,SUBLFDN
                          ST      0,LIMIT       in run-time save area
                          LH      0,I
$CHECK+                   CH      0,=H'31'
$CHECK+                   BH      SUBIFOR
                          B       COMPARE
                LOOP      LH      0,DAY
                          BCTR    0,0
                COMPARE   C       0,LIMIT
                          BL      NEXT
                          STH     0,DAY
                          statement
                          B       LOOP
                NEXT
```

6.12.5 CASE Statement

The CASE statement allows one of several possible statements to be selected and executed depending on the value of an expression. The expression may be of type SHRTINT, INTEGER, BOOLEAN, user-defined scalar, or a subrange of any of these. A particular statement is selected for execution when the value of the expression matches the case label associated with the statement. It is not necessary for the alternatives to be specified in any particular order, or even for all possibilities to be covered.

The CASE statement is of the form:

```
CASE expression OF
     case label list : statement1;
     case label list : statement2;
     - - -
     case label list : statementN
END;
```

If the expression is negative, a run-time error message is given. If the value of the expression is greater than the value (or ordinal position) of the largest (ordinally) case label, or if the expression does not match any of the case labels specified, execution continues with the statement immediately following the CASE statement.

```
1/   VAR I : INTEGER;
     - - -
        CASE I OF
           0 : statement0;
           1 : statement1;
           2 : statement2
        END;
```

```
                        L       2,I
                        CH      2,MAXLABEL
                        BH      NEXT
                        AR      2,2
                        BNM     SELECT
                        BAL     1,CASENEG       error message
SELECT                  LH      2,CASEBASE(2)
                        B       0(2,15)         index with
                                                segment base
I0          statement0
                        B       NEXT
I1          statement1
                        B       NEXT
I2          statement2
                        B       NEXT
MAXLABEL                DC      H'2'
CASEBASE                DC      H'offset of I0'  (0)
                        DC      H'offset of I1'  (1)
                        DC      H'offset of I2'  (2)
NEXT
```

2/ VAR DAY : (SUN, MON, TUES, WED, THURS, FRI, SAT);

\- - -

```
   CASE DAY OF
      MON,THURS : statement1;
      FRI,TUES  : statement2
   END;
```

```
                        L       2,DAY
                        CH      2,MAXLABEL
                        BH      NEXT
                        AR      2,2
                        BNM     SELECT
                        BAL     1,CASENEG           error message
              SELECT    LH      2,CASEBASE(2)
                        B       0(2,15)             index with
                                                    segment base
              S1        statement1
                        B       NEXT
              S2        statement2
                        B       NEXT
              MAXLABEL  DC      H'5'                ORD(FRI)
              CASEBASE  DC      H'offset of NEXT'   (SUN)
                        DC      H'offset of S1'     (MON)
                        DC      H'offset of S2'     (TUES)
                        DC      H'offset of NEXT'   (WED)
                        DC      H'offset of S1'     (THURS)
                        DC      H'offset of S2'     (FRI)
              NEXT
```

6.13 INPUT / OUTPUT

I/O is defined for 7 scalar types: SHRTINT, INTEGER, REAL, LONG REAL, BOOLEAN, CHAR and STRING. Values or addresses are passed to run-time library routines in a register, and additional information is provided with parameters stored in-line (i.e., with the code). This information is available to the library routines through the return address.

6.13.1 OUTPUT

The same output routine is called for formatted and format-free output. With format-free output a default field width is used while if a field width is specified, it replaces this default value.

In-line Parameters	Bytes	Function
TYPECODE	2	- indicates operand type
FIELDWIDTH	2	- width of field for operand
DECPLACE	2	- for reals: d in form e:f:d (digits to right of decimal point) - length of character string
WRITELNFLAG	1	- #00 - WRITE #FF - WRITELN
EFFLAG	1	- for reals only - #00 - E format #FF - F format

TYPECODE	TYPE	Default FIELDWIDTH	Register	Value/Address
1	-	-	-	-
2	SHRTINT	8	R2	value
	INTEGER	13	R2	value
3	REAL	18	F0	value
4	LONG REAL	26	F01	value
5	BOOLEAN	7	R2	value
6	CHAR	3	R2	value(character)
7	STRING(n)	n+2	R2	address

A TYPECODE of 1 indicates WRITELN without parameters. The current record is printed as it is.

```
VAR I : INTEGER;  X : REAL;  S : STRING(10);  C : CHAR;  B : BOOLEAN;
- - -
   WRITE(I);

                    L       2,I
                    L       14,APASCL011        address of
                                                output routine
                    BALR    1,14
PARMI               DC      H'2'                TYPECODE
                    DC      H'13'               FIELDWIDTH
                    DC      H'0'                DECPLACE
                    DC      X'00'               WRITELNFLAG
                    DC      X'00'               EFFLAG

NEXT
```

```
WRITELN(X:10:3, S:20, C, B:2);

                LE      0,X

                L       14,APASCL011

                BALR    1,14

PARMX           DC      H'3'            TYPECODE
                DC      H'10'           FIELDWIDTH
                DC      H'3'            DECPLACE
                DC      X'00'           WRITELNFLAG
                DC      X'FF'           EFFLAG

NEXT1           LA      2,S

                L       14,APASCL011

                BALR    1,14

PARMS           DC      H'7'            TYPECODE
                DC      H'20'           FIELDWIDTH
                DC      H'10'           DECPLACE
                DC      X'00'           WRITELNFLAG
                DC      X'00'           EFFLAG

NEXT2           IC      2,C

                L       14,APASCL011

                BALR    1,14

PARMC           DC      H'6'            TYPECODE
                DC      H'3'            FIELDWIDTH
                DC      H'0'            DECPLACE
                DC      X'00'           WRITELNFLAG
                DC      X'00'           EFFLAG

NEXT3           L       2,B

                L       14,APASCL011

                BALR    1,14

PARMB           DC      H'5'            TYPECODE
                DC      H'2'            FIELDWIDTH
                DC      H'0'            DECPLACE
                DC      X'FF'           WRITELNFLAG
                DC      X'00'           EFFLAG

NEXT4
```

6.13.2 INPUT

Different input routines are called depending on whether the input is formatted or format-free. An in-line parameter list similar to the one for output is also used.

In-line Parameters	Bytes	Function
TYPECODE	2	- indicates operand type
FIELDWIDTH	2	- width of field for formatted input - 0 for format-free input
DECPLACE	2	- for reals: d in form e:f:d (digits to right of decimal point) - length-1 of CHAR or STRING operand
READLNFLAG	1	- #00 - READ #FF - READLN
unused	1	- unused

TYPECODE	TYPE
0	-
1	SHRTINT
2	INTEGER
3	REAL
4	LONG REAL
5	BOOLEAN
6	CHAR
7	STRING

Register R0 is used to pass the address of the operand which is to receive the value read. A TYPECODE of 0 indicates READLN without parameters. In this case the remainder of the current record is disgarded.

```
VAR I : INTEGER;  X : REAL;  S : STRING(10);
- - -
   READ(I:10, X:12:4, S);
   READLN;
```

```
                    LA      0,I
                    L       14,APASCL014    formatted read
                    BALR    1,14
PARMI               DC      H'2'            TYPECODE
                    DC      H'10'           FIELDWIDTH
                    DC      H'0'            DECPLACE
                    DC      X'00'           READLNFLAG
                    DC      X'00'           unused
NEXT1               LA      0,X
                    L       14,APASCL014    formatted read
                    BALR    1,14
PARMX               DC      H'3'            TYPECODE
                    DC      H'12'           FIELDWIDTH
                    DC      H'4'            DECPLACE
                    DC      X'00'           READLNFLAG
                    DC      X'00'           unused
NEXT2               LA      0,S
                    L       14,APASCL010    format-free read
                    BALR    1,14
PARMS               DC      H'7'            TYPECODE
                    DC      H'0'            FIELDWIDTH
                    DC      H'9'            DECPLACE
                    DC      X'00'           READLNFLAG
                    DC      X'00'           unused
NEXT3               L       14,APASCL010    format-free read
                    BALR    1,14
                    DC      H'0'            TYPECODE
                    DC      H'0'            FIELDWIDTH
                    DC      H'0'            DECPLACE
                    DC      X'FF'           READLNFLAG
                    DC      X'00'           unused
```

If the variable in the input list is a numeric subrange (SHRTINT or INTEGER) and $CHECK+ is specified, the value read is checked to ensure that it is in the correct range. The input routine assigns the value to the variable but on return to the calling program, makes the assigned value available in R0. This allows in-line code to be generated to perform the necessary bounds checks.

```
VAR I : 20..30;
- - -
   READLN(I:10);
```

```
                    LA      0,I
                    L       14,APASCL014    formatted read
                    BALR    1,14
PARMI               DC      H'2'            TYPECODE
                    DC      H'10'           FIELDWIDTH
                    DC      H'0'            DECPLACE
                    DC      X'FF'           READLNFLAG
                    DC      X'00'           unused
NEXT                C       0,=F'30'        high bound
                    BAL     1,SUBHRD        check for high value
                    C       0,=F'20'        low bound
                    BL      SUBLRD          error message (low)
NEXT1
```

6.14 PROCEDURE / FUNCTION CALLS

The calling sequences for procedures and functions are identical. Upon return a function returns a quantity (value or address) in a register which is then used in the subsequent code.

The simplest subprogram call is for a procedure without parameters as in:

```
SUM
                    L       15,base address of SUM
                    BALR    14,15
                    L       15,base address of current segment
```

If parameters are passed to a subprogram, a list of addresses is established and R1 is loaded with the address of this list. The information passed is identical for "call by reference" and "call by value" parameters. The only difference is in the way the called routine uses the information passed to it.

Formal Parameter	Parameter List Entry
SHRTINT INTEGER REAL LONG REAL BOOLEAN CHAR STRING SET SCALAR (user-defined) SUBRANGE (of scalar type) POINTER	Address of location containing value
ARRAY RECORD	Base address of array or record
PROCEDURE FUNCTION	Address of location containing address of procedure or function

All simple variables which are passed "by reference" are moved into locations in the current data segment on entry, and this copy is used in all references to the variable. This is done to eliminate an overhead which would otherwise be imposed in accessing these variables. The final values of these variables passed "by reference" are copied back to the original variable locations on exit from the subprogram. These parameters are really handled with a "call by result" rather than a "call by reference" mechanism.

Arrays and records passed "by reference" are accessed directly. If structured types are passed "by value", space for a copy is acquired at entry and the entire structured type is copied into this block of memory. This copying is accomplished by a library routine (PASCMOVE), which accepts the address of the original in R3, the address of the new location in R2, and the length to be moved as an in-line parameter. The entire structured type is copied with MVCs rather than element by element.

If a procedure or function is passed as a parameter, then the address in the parameter list is used in any subsequent calls of this subprogram, rather than setting up an ESD and RLD entry to get the address.

Function result values are returned to the calling program in a register according to the following table:

Result Type	Register
SHRTINT	R0
INTEGER	R0
REAL	F0
LONG REAL	F01
BOOLEAN	R0
CHAR	R0
SET	R0
SCALAR (user-defined)	R0
SUBRANGE (of scalar type)	R0
POINTER	R0

R0 is used to return all but real values to conform to IBM conventions. R0 and R1 are universal work registers and therefore are never left holding a value which is entered on the operand stack. For this reason, the code for a function call returning a value in R0 is immediately followed with the instruction:

```
LR      2,0
```

and an entry with R2 is made on the operand stack.

```
TYPE VECTOR = ARRAY(1..10) OF REAL;
VAR Y,Z : REAL;  I : INTEGER;  X : VECTOR;
   PROCEDURE TEST;
   BEGIN - - - END;
   FUNCTION SUM (VAR A : VECTOR;  J : INTEGER;  PROCEDURE P) : REAL;
   BEGIN - - - END;
BEGIN
   - - -
   Y := Z + SUM(X, I, TEST);
```

```
L       1,base address of array X
ST      1,PARM1          in run-time save area
LA      1,I
ST      1,PARM2          in run-time save area
LA      1,address of address of TEST
O       1,=X'80000000'   Sign bit(last parm)
ST      1,PARM3          in run-time save area
LA      2,save area in PASCALDS
LA      1,PARM1          address of parm list
L       15,base address of SUM
BALR    14,15
L       15,base address of main program
AE      0,Z
STE     0,Y
```

If SUM is called from another subprogram rather than from the main program, then the statement

```
LA      2,save area in PASCALDS
```

is replaced by

```
LR      2,n
```

where n is the base register of the current segment data area. This data area contains a save area for any other PASCAL routines which are called from this subprogram. The called routine has no way of knowing the declaration level of the calling routine (each level uses a different register as the base of the data area), and therefore is passed the necessary address in R2.

6.15 ARRAYS

Arrays are stored by rows. The declaration

```
VAR X : ARRAY(1..10,10..20,BOOLEAN) OF INTEGER;
```

is equivalent to:

```
VAR X : ARRAY(1..10) OF ARRAY(10..20) OF ARRAY(BOOLEAN) OF INTEGER;
```

Dope vector information for arrays is not necessarily grouped together into a common table because the information may be processed at widely varying times due to TYPE declarations. The following information is available at run time:

- base address of the entire array
- the component size of each dimension of the array
- the upper and lower bounds of each index, if it is a subrange.

The component sizes are restricted to a maximum of 32767 bytes so that the MH instruction can be used. The size of the entire array is not restricted.

Example:

```
VAR X : ARRAY(1..10,1..75,1..100) OF INTEGER;
```

The component size of:

X(1,1,1) is 4 bytes (i.e., size of INTEGER)

X(1,1) is 100 * 4 (i.e., size of X(1,1,1)) = 400 bytes

X(1) is 75 * 400 (i.e., size of X(1,1)) = 30000 bytes

none of which is greater than 32767 bytes. The size of the entire array X is 10 * 30000 (i.e., size of X(1)) = 300000 bytes.

If an array component type is a simple type, then shifts are performed instead of multiplications where possible:

Shift Left (bits)	Simple Type
1	SHRTINT
2	INTEGER, REAL, BOOLEAN, SET, SCALAR, POINTER
3	LONG REAL

A subtract instruction is saved on an index calculation if the index is of type BOOLEAN, user-defined scalar or a subrange of integers where the lower bound is 0. If a constant is used as an array subscript, it is checked for validity at compile time. If the constant value, minus the lower bound for this particular subscript, times the component size, is less than 4096, a LA instruction is used to update the accumulating address rather than the normal subscripting.

Run-time checking code is generated to check array subscripts when the index type is a subrange of integers, if $CHECK+ is specified. Indexes of type BOOLEAN or user-defined scalar are checked for validity at compile time, eliminating the need for run-time checks in these cases.

Some examples are shown with $CHECK+ specified. The extra instructions which are generated to perform the checking are indicated. If these indicated instructions are removed, the resulting code sequence is what is generated with $CHECK-.

```
1/   VAR I : INTEGER;  J : SHORT INTEGER;  Z : REAL;
       X : ARRAY(1..10,1..5) OF REAL;
     - - -
       Z := X(I,J);
```

Checking Code

```
                    L       11,base address of X
                    L       1,I
$CHECK+             C       1,=F'10'        upper bound
$CHECK+             BAL     1,ARRAYERR      check high bound
                    S       1,=F'1'         lower bound
$CHECK+             BM      MAINERR         error message (low)
                    MH      1,=H'20'        component size of X(I)
                    AR      11,1            update address
                    LH      1,J
$CHECK+             C       1,=F'5'         upper bound
$CHECK+             BAL     1,ARRAYERR      check high bound
                    S       1,=F'1'         lower bound
$CHECK+             BM      MAINERR         error message (low)
                    SLA     1,2             multiply by 4 (REAL)
                    AR      11,1            update address
                    LE      0,0(,11)        REAL value X(I,J)
                    STE     0,Z
```

2/ VAR I : INTEGER; J : SHORT INTEGER; Z : REAL;
X : ARRAY(0..10,0..5) OF REAL;
- - -
Z := X(I,J);

```
L       11,base address of X
L       1,I
MH      1,=H'24'        component size of X(I)
AR      11,1            update address
LH      1,J
SLA     1,2             multiply by 4 (REAL)
AR      11,1            update address
LE      0,0(,11)        REAL value X(I,J)
STE     0,Z
```

This is the same example as 1/ with two changes. $CHECK- was specified which eliminated the bounds checks on the subscripts. Also, the lower bounds of the array subscripts were set to zero, eliminating a subtract instruction.

If the array subscripts are constants, further savings in code are achieved.

3/ Declarations as above
Z := X(3,0);

```
L       11,base address of X
LA      11,72(11)       address of X(3)
                        also of X(3,0)
LE      0,0(,11)
STE     0,Z
```

```
4/   TYPE WEEK = (SUN, MON, TUES, WED, THURS, FRI, SAT);
        TOY = (CAR, BOAT, PLANE);
        WKWEEK = MON..FRI;
     VAR WK : WKWEEK;
         B  : BOOLEAN;
         S  : STRING(5);
         MAT : ARRAY(TOY, BOOLEAN, WKWEEK) OF STRING(10);
     - - -
        S := MAT(BOAT, B, WK);
```

```
L       11,base address of MAT
LA      11,100(,11)       address of MAT(BOAT)
L       1,B
MH      1,=H'50'          component size of
                          MAT(BOAT,B)
AR      11,1              update address
L       1,WK
S       1,=F'1'           ORD(MON)
MH      1,=H'10'          size of STRING(10)
AR      11,1              update address
MVC     S(5),0(11)        assign truncated string
```

The code generated for this statement is the same for $CHECK+ and $CHECK- as all the required range checks are performed at compile time.

6.16 RECORDS

The only information needed at run time is the base address of the record. Information on the size of record fields, and the locations of fields within records is processed at compile time and is incorporated in the code which is generated. The compiler imposes the restriction that no record field can start more than 32767 bytes from the start of the record.

If a record field is specified by qualifying the record name, the base address of the record is loaded into a register, and then the address is updated to indicate the particular field with a LA instruction (using an index register if necessary). PASCALDS contains a table of constants of multiples of 4K which are used for these index values if necessary.

```
1/   VAR X : RECORD
                N : RECORD
                      A, B : REAL;
                    END;
                A : ARRAY(1..150,1..10) OF REAL;
                B : REAL
              END;
           Y : REAL;
     - - -

1a/      Y := X.N.A;

                              L       11,base address of X (also X.N.A)
                              LE      0,0(,11)
                              STE     0,Y
```

1b/ X.N.B := X.B;

```
L       11,base address of X
LA      11,4(,11)          address of X.N.B
L       10,base address of X
LH      1,=H'4096'
LE      6,1912(1,10)       value of X.B
STE     6,0(,11)           location of X.N.B
```

Structured types may be nested. The addressing mechanism remains exactly the same as for simpler structured types.

```
2/  VAR X : ARRAY(1..10) OF RECORD
                              I : INTEGER;
                              A : RECORD
                                    R : LONG REAL;
                                    B : ARRAY(BOOLEAN) OF RECORD
                                                            X : LONG REAL;
                                                            Y : INTEGER
                                                          END
                                  END
                            END;
        K : INTEGER;  P : BOOLEAN;
```

- - -

2a/ X(1).I := X(K).A.B(P).Y;

```
L       11,base address of X (also X(1).I)
L       10,base address of X
L       1,K
S       1,=F'1'             lower bound
MH      1,component size
AR      10,1                address of X(K)
LA      10,4(,10)           address of X(K).A
LA      10,8(,10)           address of X(K).A.B
L       1,P
MH      1,=H'12'            component size
AR      10,1                address of X(K).A.B(P)
L       10,4(,10)           value of X(K).A.B(P).Y
ST      10,0(,11)           location X(1).I
```

WITH Statement

The WITH statement opens the scope to a record so that the record fields can be accessed directly, without having to qualify the record variable name. The address of the record variable in the WITH statement is computed and then stored in the run-time save area. When a record field is specified inside the WITH statement, this address is recovered and then the field address is determined in the usual manner as an offset from this record base address.

```
1/   VAR X : ARRAY(1..10) OF RECORD
                              X : LONG REAL;
                              N : RECORD
                                    NAME, FIRST : STRING(10);
                                    K : INTEGER;
                                    BIRTH : RECORD
                                              DAY : 1..31;
                                              MONTH : 1..12;
                                              YEAR : 1850..2000
                                            END
                                END
                            END;
         I : INTEGER;
     - - -

     WITH X(I).N DO BEGIN

                              L       11,base address of X

                              L       1,I

                              S       1,=F'1'          low bound

                              MH      1,component size

                              AR      11,1             address of X(I)

                              LA      11,8(,11)        address of X(I).N

                              ST      11,TWITH1        in run-time save area

        WITH BIRTH DO BEGIN

                              L       11,TWITH1        address of X(I).N

                              LA      11,24(,11)       address of X(I).N.BIRTH

                              ST      11,TWITH2        in run-time save area

           K := YEAR;

                              L       11,TWITH1        address of X(I).N

                              LA      11,20(,11)       address of X(I).N.K

                              L       10,TWITH2        address of X(I).N.BIRTH

                              L       10,8(,10)        value of X(I).N.BIRTH.YEAR

                              ST      10,0(,11)        location  X(I).N.K
```

6.17 POINTERS

Pointers occupy a full-word of memory. Each pointer variable is associated with a TYPE identifier. The value of a pointer variable is the address of an occurrence of the associated type, or NIL if the pointer variable is not currently associated with an occurrence.

The dynamic variable indicated by a pointer is referenced by qualifying the pointer variable with an @ symbol.

Variables are allocated dynamically through the use of the procedure NEW. The address of the pointer variable is passed to NEW and the size of the dynamic variable to be generated is passed as an in-line parameter. Procedure NEW allocates a block of memory of the required size and stores the memory address in the pointer variable.

```
1/   TYPE DATE = RECORD
                   DAY : 1..31;
                   MONTH : 1..12;
                   YEAR : INTEGER;
                   AD : BOOLEAN
                 END;
     VAR P, Q : @DATE;
         I : INTEGER;
     - - -

        NEW(P);
```

```
                              LA      2,P
                              L       0,current free core pointer for proc/func
                                            allows check with core top limit
                              L       14,APASCL008      dynamic core routine
                              BALR    1,14              stores address in P
                              DC      F'16'             size of DATE in bytes
                                                        NOT full-word aligned
```

```
        Q := P;
```

```
                              L       1,P
                              ST      1,Q
```

```
     I := Q@.YEAR;
```

```
                              L       11,Q              address of occurrence
                                                        of DATE
                              L       11,8(,11)         value Q@.YEAR
                              ST      11,I
```

6.18 ASSIGNMENTS

Most possible assignments have been demonstrated in the preceding examples. The following describes the cases which have not been discussed.

Strings

If the string on the left is shorter than the string on the right, the assignment is made by truncating the long string on the right. If the string on the left is longer than the string on the right, the string is padded out to the required size with blanks on the right.

```
VAR S : STRING(10);
    T : STRING(20);
- - -
   S := T;
                          MVC     S(10),T

   T := S;
                          MVI     T+10,C' '
                          MVC     T+11(9),T+10
                          MVC     T(10),S
```

Subranges

Run-time checking code for numeric subranges has been demonstrated for most cases. When the variable being assigned a value is a subrange of integers and $CHECK+ is specified, run-time checking code is produced to guarantee that the assigned value is within the accepted limits.

```
VAR I : INTEGER;
    K : -10..10;
- - -
   K := I;
```

Checking Code

```
                    L       0,I
$CHECK+             C       0,=F'10'        upper bound
$CHECK+             BAL     1,SUBERR        check upper bound
$CHECK+             C       0,=F'-10'       lower bound
$CHECK+             BL      SUBMERR         error message (low)
                    ST      0,K
```

The check requires 4 extra instructions in-line and the execution of 5 extra instructions to ensure that the assignment is valid.

The compiler performs range checks when constants are assigned to subranges if $CHECK+ is specified. For this reason, the following optimizations are possible.

```
K := 3;
                    LA      1,3
                    ST      1,K

K := 0;
                    SR      1,1
                    ST      1,K
```

Structured Types

Assignment of structured types is allowed if the type descriptors are identical. The same run-time library routine (PASCMOVE) is used to copy the information as is used when structured types are passed "by value" into subprograms. The copying is accomplished with MVC instructions, rather than element by element, or field by field.

```
1/   VAR X, Y : ARRAY(1..10,1..10) OF INTEGER;
         M, N : RECORD
                   A : INTEGER;
                   B : REAL;
                   C : ARRAY(1..10) OF REAL
                END;
     - - -

1a/      X := Y;
```

```
                    L       2,base address of X
                    L       3,base address of Y
                    L       15,APASCMOVE      address of copy routine
                    BALR    14,15
                    DC      F'400'            size in bytes
                                              NOT full-word aligned
                    L       15,base address of current segment
```

```
1b/     M := N;

                        L       2,base address of M
                        L       3,base address of N
                        L       15,APASCMOVE      address of copy routine
                        BALR    14,15
                        DC      F'48'             size in bytes
                                                  NOT full-word aligned
                        L       15,base address of current segment

1c/     X(3) := Y(5);

                        L       11,base address of X
                        LA      11,80(,11)        address of X(3)
                        L       10,base address of Y
                        LA      10,160(,10)       address of Y(5)
                        LR      2,11
                        LR      3,10
                        L       15,APASCMOVE      address of copy routine
                        BALR    14,15
                        DC      F'40'             size in bytes
                                                  NOT full-word aligned
                        L       15,base address of current segment
```

CHAPTER 7

CONCLUSIONS AND DIRECTIONS FOR FURTHER WORK

7.1 CONCLUSIONS

The principal result of the work is that a production compiler for PASCAL, implemented on IBM 360/370 machines, has been produced. This compiler, completed at this time, seems to be in a position to have a major impact on the widespread use and acceptance of the language PASCAL. Lecarme in his article "Structured Programming, Programming Teaching and The Language Pascal" (Lec74a) in July 1974, stated "No useful implementation is presently available on an IBM machine". It is hoped that this compiler will fill this need.

Besides producing a working PASCAL compiler, several other points have been demonstrated. The approach used is novel in two ways. This is the only PASCAL compiler for IBM machines which is not based on the original Zurich compiler for CDC machines, and is one of the few compilers written with a general purpose translator-writing system. The results show that this approach is entirely feasible, if the language meets the requirements of having a straightforward and well-defined syntax. The most notable other example of the use of a translator-writing system on IBM machines is XPL(McK70).

It has been shown that it is possible to take advantage of the redundancy of the language, especially in the definition of user-defined scalars and subranges thereof, to perform most necessary

checks at compile time and thus eliminate the need for expensive run-time checking code. The compiler demonstrates that it is possible, in one pass, to produce code for complex logical expressions which is as efficient as that produced by many multi-pass compilers. This optimization, and the approach of making all branches direct, are greatly aided by the decision to restrict all segments to 4K bytes, and demonstrates a strong reason for keeping segments small and manageable, especially on a machine of this architecture.

In the area of compiler organization, the technique of running all the internal tables from displays, greatly eases the work necessary to make minor changes, such as adding an extra field to an entry. This approach allows simple adjustment of table sizes, as is done with the SETLIMIT routine or parameters passed on the EXEC card. The idea of a compiler initialization routine, of which SETLIMIT is a primitive example, offers the advantage of tailoring a compiler to suit the needs of an installation without searching through the source of the compiler in an attempt to find the declarations which need modifying.

The compiler demonstrates a forgiving approach to simple, often-made syntax errors. Such things as an equal sign (=) after VAR or a colon (:) after TYPE are accepted with a suitable warning message. A semi-colon (;) before an END in a RECORD or CASE statement, are likewise handled. Other common errors such as a semi-colon (;) before ELSE, or after the final END of the program, are not sufficient to prevent successful compilation. Reserved words which are actually

two words are allowed as either one or two words (eg. LONGREAL or LONG REAL and GOTO or GO TO). The points demonstrated are errors which are often made even by experienced programmers. As long as the compiler can determine what is required without ambiguity, the effort should be made. The top-down approach is well suited to this task.

Richmond (Ric75) states:

> "Documentation for Pascal is sparse, particularly concerning the internal construction of the compiler. The only published internal documentation on the compiler are the original papers on the design of the compiler (Wir71b) and on code generation (Wir72b)."

This thesis therefore represents one of the most complete descriptions of the design strategy and internal workings of any PASCAL compiler, and in particular, any for IBM machines.

7.2 EVALUATION OF THE APPROACH

The SYNTICS system performed well, although when work on the compiler was begun, it was in a primitive state. Many of the SYNTICS support routines were heavily modified for the PASCAL compiler. The main drawback to SYNTICS at this time is the sparse documentation (Kos72) although further documentation may become available (Kos75).

The PASCAL compiler is presently the only compiler completed using the SYNTICS system and is therefore the only demonstration that the approach is feasible. The SYNTICS system could become a widely used general purpose translator-writing system with better documentation and some further work. Its use by others is not recommended at this

time unless they have access to the source code and can consult with one of the two people familiar with the system.

SYNTICS is a PL360 based system and this makes the use of PL360 the natural choice for the compiler. The modular nature of the compiler, with approximately 250 procedures, is feasible with PL360 because of the low overhead on procedure calls, although this might become very expensive in a high-level language. The primitive data structures and lack of dynamic arrays did not cause any problems. No array subscripting was performed in the compiler. Access to tables was with direct pointers or through the use of an index register.

The most frustrating feature of PL360 was the small symbol table size in the version of the compiler which was available. PL360 does not check for table overflows and the addition of a single declaration can cause meaningless error messages. The body of the PASCAL compiler had to be broken into four segments to be compiled and then later linked together. The restriction that procedure names had to be declared before they were used caused considerable organizational problems when performing split compilations.

Overall the approach used was highly successful. Considering the vast library of routines which has been developed, and the experience which has been gained with this strategy, this approach would be the author's first choice in any future projects of a similar nature.

7.3 FURTHER WORK

The most obvious area for further work is the implementation of a complete file-handling system, rather than just restricting I/O to the standard input and output files. This would greatly extend the versatility and appeal of the compiler. This complete file-handling system was not included in the initial version, to keep the amount of work, and time involved, within reasonable limits.

At present the compiler is neither reusable nor reentrant. PL360 does not lend itself to writing reentrant programs, but it is intended to make the compiler reusable in the future.

At present, no garbage collection is performed. This remains a low priority project because in the foreseeable future, the time could be better spent implementing the constructs described above.

A method of allowing compile-time initialization, at least for the main program, is being considered. This initialization is not currently defined in the language, although several implementations have accomplished this with a separate VALUE part in the declarations for the main program.

Work on the project has helped to point out some deficiencies in PASCAL. Several constructs in the language are very error prone, and in future languages, or in a later version of PASCAL, some of these deficiencies should be corrected.

Procedures and functions as formal parameters are a source of problems because parameter number and type checking cannot be performed at compile time on any subsequent calls of these routines.

One obvious solution is to require that the procedure or function parameters be specified immediately after the name in the formal parameter list. An attempt has been made to provide a solution to this problem by allowing the FORWARD declaration to announce the parameters of procedures and functions which are used as formal parameters.

Pointers allow a great deal of power in manipulating complex data structures, but this is at the expense of side-stepping compile-time checks. Even if a data structure is passed into a routine "by value", there is no way of preventing the modification of "supposedly inaccessible" constructs through the use of pointers. A garbage collector, if implemented, must be non-trivial to detect that allocations are no longer accessible, if parts of complex chains are released using DISPOSE.

Global labels complicate the language and the whole handling of labels, and therefore are not supported in this implementation. If global labels were eliminated from the language, the requirement that all labels must be declared could be removed.

The variant part of records are error-prone unless a run-time check on the tagfield value is made every time a field in the variant part is accessed. This variant part is the only means of accomplishing equivalencing, and this feature would be destroyed if run-time checks were performed.

Although there are some deficiencies and drawbacks in PASCAL, it is still a major step in a new direction. It provides power with

a simple design. It introduces new data types and a straightforward means of defining complex data structures. The number of machines on which it has been implemented demonstrates that it is not tied too strongly to the architecture of any particular machine. The massive effort to implement the language on all these machines indicates the widespread appeal of the language. Interest in the language is steadily growing and indications are that it will become one of the major programming languages in the near future. It is hoped that this thesis will further that end.

BIBLIOGRAPHY

Bau68a Bauer, H.R., Becker, S., Graham, S.L., "ALGOL W Language Description", Report CS 89, Computer Science Department, Stanford Universiry (March 1968)

Bau68b Bauer, H., Becker, S., Graham, S., "ALGOL W IMPLEMENTATION", Report CS 98, Computer Science Department, Stanford University (May 1968)

Dev74 Deverill, R.S., Hartmann, A.C., "Interpretive Pascal for the IBM 370", Information Science Technical Report No. 6, California Institute of Technology (August 1974)

Fou72 Foulkes, W.B., Kossmann, R., "The DECLAB Logic Language", Department of Computer Science, University of Manitoba (April 1972)

Fou73 Foulkes, W.B., Wells, J.M., "A Pascal Compiler for IBM 360/370 Computers", Proceedings of Third Manitoba Conference on Numerical Mathematics, pp 427-451 (October 1973)

Gri71 Gries, D., "Compiler Construction for Digital Computers", John Wiley & Sons Inc., New York (1971)

Gau75 Guarino, L., Private Communication, Computation Research Group, Stanford University, Stanford, California (July 1975)

Hab73 Habermann, A.N., "Critical Comments on the Programming Language Pascal", ACTA INFORMATICA 3, pp. 47-57 (1973)

Har74 Hartmann, A., Private Communication, California Institute of Technology, Pasadena, California (1974)

Hoa73 Hoare, C.A.R., Wirth, N., "An Axiomatic Definition of the Programming Language Pascal", ACTA INFORMATICA 3, pp. 335-355 (1973)

IBMa IBM, "System/360 Operating System Assembler Language", IBM Form C28-6514

IBMb IBM, "System/360 Operating System Assembler(F) Programmers Guide", IBM Form GC26-3756

IBMc IBM, "System/360 Principles of Operation", IBM Form A22-6821

IBMd IBM, "System/360 Operating System FORTRAN IV (H) Compiler", IBM Form Y28-6642

Jen74 Jensen, K., Wirth, N., "Pascal User Manual and Report", Lecture Notes In Computer Science, Vol. 18, Springer-Verlag, New York (1974)

Kos72 Kossmann, R., "The RSYN and SYNTICS T. W. S.", Department of Computer Science, University of Manitoba (June 1972)

Kos75 Kossmann, R., Ph.D. Thesis in Progress, Department of Computer Science, University of Manitoba

Lar75 Larmouth, J., Private Communication, Computer Laboratory, University of Cambridge, England (August 1975)

Lec74a Lecarme, O., "Structured Programming, Programming Teaching, and the Language Pascal", SIGPLAN NOTICES 9,7, pp. 15-21 (July 1974)

Lec74b Lecarme, O., Desjardins, P., "Reply to a Paper by A. N. Habermann on the Programming Language Pascal", SIGPLAN NOTICES 9, 10, pp. 21-27 (October 1974)

Mal71 Malcolm, M.A., "PL360 (Revised) A Programming Language For The IBM 360", STAN-CS-71-215, Computer Science Department, Stanford University (May 1971)

McK70 McKeeman, W.M., Horning, J.J., Wortman, D.B., "A Compiler Generator", Prentice-Hall, Englewood Cliffs (1970)

Moi71 Moir, D.A., "Format-Directed Input/Output for Algol W", M.Sc. Thesis, Department of Computer Science, University of Manitoba (January 1971)

Ric74a Richmond, G., "Pascal Newsletter", No. 1, University of Colorado Computing Center, Boulder (January 1974)

Ric74b Richmond, G., "Pascal Newsletter", No. 2, University of Colorado Computing Center, Boulder (May 1974)

Ric75 Richmond, G., "Pascal Newsletter", No. 3, University of Colorado Computing Center, Boulder (February 1975)

Rus74 Russell, D.L., Sue, J.Y., "STANFORD PASCAL360 Implementation Guide", Technical Memo 89, Stanford University, Stanford, California (November 1974)

Tas Tassart, P., Private Communication, I.R.E.P., Grenoble, France

Wel72 Welsh, J., Quinn, C., "A Pascal Compiler for the ICL 1900 Series Computers", SOFTWARE-PRACTICE AND EXPERIENCE 2, 1, pp. 73-77 (1972)

Wir66 Wirth, N., Hoare, C.A.R., "A Contribution to the Development of Algol", COMMUNICATIONS OF THE ACM 9, 6, pp. 413-432 (1966)

Wir68 Wirth, N., Wells, J.W., Satterthwaite, E.H., "The PL360 System", Technical Report No. CS 91, Computer Science Department, Stanford University (April 1968)

Wir71a Wirth, N., "The Programming Language PASCAL", ACTA INFORMATICA 1, 1, pp. 35-63 (1971)

Wir71b Wirth, N., "The Design of a Pascal Compiler", SOFTWARE-PRACTICE AND EXPERIENCE 1, 4, pp. 309-333 (1971)

Wir72a Wirth, N., "The Programming Language Pascal (Revised Report)", No. 5, Berichte der Fachgruppe Computer-Wissenschaften, Eidgenössische Technische Hochschule, Zürich (November 1972)

Wir72b Wirth, N., "On Pascal, Code Generation, and the CDC 6400 Computer", Computer Science Department, STAN-CS-72-257, Stanford University (1972) (out of print, Clearinghouse stock no. PB208519)

Wir73 Wirth, N., "Systematic Programming: an Introduction", Prentice-Hall, Englewood Cliffs (1973)

Note: Lec74a contains an extensive bibliography.