IMPLEMENTATION OF A SUBSET OF MODES IN AN ALGOL 68 COMPILER

By

WALTER MICHAEL SEAY

Bachelor of Science

Troy State University

Troy, Alabama

1974

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1976

IMPLEMENTATION OF A SUBSET OF MODES IN AN ALGOL 68 COMPILER

Thesis Approved:

R. E. Hedrick
Thesis Adviser

James R. VanDoren

Eugene Bailey

Norman N. Durham
Dean of Graduate College

PREFACE

This thesis is a description of the mode facilities which have been added to the Oklahoma State University ALGOL 68 Compiler. Also included is a description of the changes that were required to update the language accepted by the compiler in accordance with the newest definition.

I would like to thank the faculty of the Computing and Information Sciences Department for their assistance and their desire to teach. A special thanks is in order to my advisor, Dr. G. E. Hedrick, for his invaluable assistance and understanding during my stay at Oklahoma State University. I would also like to thank my two sons, Bobby and Johnny, who often were required to be quieter than little boys should ever have to be. It is impossible for me to express properly my thanks to my wife, Kathy, who did so much more than type.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Objectives

Since 1973 a project has been underway at Oklahoma State University to write a portable compiler for the ALGOL 68 language (1) (2) (3). This very powerful programming language gives the programmer the capability of defining and using his own set of data types in addition to a predefined set. The treatment of data types and values of these data types has been formalized in ALGOL 68 to the concept of a mode (4) (5).

Prior to 1976 the Oklahoma State University ALGOL 68 Compiler had the capability to recognize a very limited set of modes. One objective of this thesis is to show how a greater number of modes can be accepted while allowing the compiler to remain within the (size and portability) constraints which have been placed on the compiler by its original implementer (1). Another objective of this thesis is to detail some changes necessary to allow the Oklahoma State University ALGOL 68 Compiler to conform to the language defined in the "Revised Report on the Algorithmic Language ALGOL 68" (5), rather than the original ALGOL 68 language

defined in the "Report on the Algorithmic Language ALGOL 68" (4).

It is assumed that the reader is familiar with the basic rules of ALGOL 68 and has some familiarity with the terminology. Appendix A contains a glossary of terms which are used in this thesis in order to facilitate its reading. Familiarity is assumed with the features implemented in the Oklahoma State University ALGOL 68 Compiler also. If the reader has a limited knowledge of the compiler's features, then John Jensen (1) is the best source to obtain the necessary background information. The thesis by Alan Eyler (3) may also be helpful.

History of the Oklahoma State University ALGOL 68 Compiler

The Oklahoma State University ALGOL 68 Compiler originally was implemented as a master's thesis by Jensen (1) in July of 1973. The original ALGOL 68 subset compiler was a scientific subset of ALGOL 68. A major design criterion was to develop a portable compiler; in order to achieve this goal, the compiler was written using IBM 1130 Basic FORTRAN. The compiler was implemented on an IBM 1130 computer with 8K 16-bit words of storage, a card reader/punch, and a console typewriter as the principal output device. The compiler also uses a single disk storage device for storage of intermediate code and simulated program memory. In order to insure portability, the code generated from the compiler

is "machine code" for a pseudo-machine which is then interpretively executed by a FORTRAN program. The small size of the IBM 1130 computer, while enhancing the portability characteristics of the compiler, restricted the set of features which could be implemented by Jensen.

At the same time Jensen was implementing the Oklahoma State University ALGOL 68 Compiler, Roger Berry (2) completed his master's thesis regarding the implementation of formatted transput. ALGOL 68 formatted transput is an extremely sophisticated and versatile input/output formatting package. Berry's implementation is a package capable of interpretive execution independent of any ALGOL 68 compiler.

Berry's (2) formatted transput package is in the process of being incorporated with the Oklahoma State University ALGOL 68 Compiler on the IBM 360/65 computer system. The combined system will allow the ALGOL 68 programmer to use the formatted input/output package directly. The combined version will not execute on the IBM 1130 due to its large size and due to the use of full standard FORTRAN in the transput package.

In the spring of 1975, Eyler (3) completed implementation of procedures for the IBM 1130 version of the compiler. The procedure facility which he implemented allows procedure constants--a facility approximately equivalent to ALGOL 60 or PL/I procedures. It supports recursive procedure invocations.

Several students have volunteered time to improve the original compiler of Jensen. Major work has been accomplished by these people. This work includes implementation of the CASE statement, rewriting the file handling capability (standard input and standard output files only) of the compiler, and now the incorporation of Mr. Berry's transput package as an integral part of the compiler.

There are currently several versions of the Oklahoma State University compiler; the IBM 1130 version with procedures, the IBM 360 version without procedures, and the IBM 360 version with procedures. These versions also are operational on the TI ASC computer and the XDS Sigma 5 computer. Currently work is under way to provide a single version on the IBM 360/65 which includes formatted transput and procedures. When this version is completed, the result will be an extremely versatile and powerful programming tool.

Literature Review

The ALGOL 68 language is defined in the "Revised Report on the Algorimthic Language ALGOL 68" (5). Two good books which survey the ALGOL 68 language are an ALGOL 68 Companion (6) and An Informal Introduction to ALGOL 68 (7). Of these two, the ALGOL 68 Companion is the easier to comprehend. Another excellent source of information is the highly readable ALGOL 68-R Users Guide (8). This users guide introduces the basic language features without introducing

much of the new terminology found in the other documents mentioned above.

Information about the Oklahoma State University ALGOL 68 subset compiler can be found in the master's theses by Jensen (1), Berry (2) and Eyler (3). Details concerning other implementations of ALGOL 68 can be found in proceedings of several conferences held for ALGOL 68 implementers. For example, the proceedings edited by J. E. L. Peck entitled ALGOL 68 Implementation (9) contains a description of one of the most successful production compilers of ALGOL 68--ALGOL 68-R. The ALGOL 68-R compiler was produced for the Royal Radar Establishment, Malvern, England. It contains many of the features of the full language and it is used as the primary programming language at the Royal Radar Establishment. Descriptions of several other operational (and almost operational) compilers can be found in the Proceedings of the 1975 International Conference on ALGOL 68 (10).

Much of the literature which has been written about ALGOL 68 has been concerned with the treatment of modes. Many of the methods which implement full ALGOL 68 modes require complex storage structures for their representation and also require considerable processing time.

Peck (11) suggested that an ALGOL 68 mode could be represented by a Greibach Normal Form Grammar. The dissertation by Mary Zosel (12) utilized the grammatical representation of modes to develop algorithms for equivalencing, coercion, balancing and operator identification in an

ALGOL 68 program. The methods developed by Zosel provide a comprehensive treatment of modes; they are, however, difficult to implement in FORTRAN due to FORTRAN's lack of recursive procedures and list processing facilities. The algorithms presented by Zosel are based upon the original report which specified a slightly different treatment of modes than that specified by the revised report. This thesis is based on the revised report (5).

J. Král (13) shows that ALGOL 68 modes can be represented by a finite automaton. This allows an implementer to use the existing algorithms for manipulating finite automata upon ALGOL 68 modes, such as reducing the automaton (mode) to a canonical representation (i.e., equivalencing modes).

H. J. Lane (14) presents methods which allow coercion sequences to be determined by using boolean matrix techniques upon modes which have been represented in grammar form. The amount of storage required for these matrices can be quite large if the number of modes is large.

This thesis specifies how a limited (but useful) mode facility can be implemented in a portable compiler with less overhead than a full mode implementation would require.

CHAPTER II

ALGOL 68 MODES

Introduction

Most higher level programming languages embrace the concept of data type. A data type names a class of values which may be represented in the machine (either by the hardware or by software implementation). For example, FORTRAN allows a variable declared with the integral data type (e.g., INTEGER X) to possess positive or negative integral values. Some programming languages allow the programmer to define structures; structures are aggregates of other predefined data types. PL/I and COBOL for example, allow structures to be declared. Both languages provide mechanisms for manipulating a structure as an aggregate and also provide for manipulating the individual elements (15) (16).

ALGOL 68 has generalized the concept of data type. This generalization is the concept of mode. There are five basic modes in ALGOL 68: BOOL (boolean), INT (integral), REAL (floating point), FORMAT, and CHAR (character). The programmer may construct new modes using the notions of row, reference-to, procedure, union, and structure (these are defined below). In full language implementations of ALGOL 68 the programmer is allowed to apply the notions (row,

reference-to, etc.) to modes which he has previously defined to form more intricate modes.

Tools for Building New Modes

The notion row may be applied to a mode to obtain a new mode which specifies a multiple set of values of the old mode. The row notion is displayed by the square brackets ([]). Since INT is a basic mode which specifies an integral value, then [] INT specifies a multiple of integral values (commonly called a vector). Values of a row-ed mode may be indexed to obtain a single value of the mode or sliced to obtain a subset multiple of the original set of values.

A mode (such as REAL) may be preceded by the symbol REF to form a new mode REF REAL (read reference-to-real mode). When an object does not have the REF symbol as the first symbol of its mode, then that object is a value of the mode; e.g., 3.5 is of mode REAL. If an object has the form REF amode (where amode is a user defined mode or a basic mode), then that object is a name (address) which may refer to a value of the mode amode. An object of mode REF amode is usually called an amode variable since it performs the same function as a variable in other programming languages. If the mode of an object has the form REF REF amode, then the object is similar to a PL/I pointer variable; that is, the object may reference (point to) a <u>variable</u> of mode amode. It is possible for an object which has the form REF^n amode

(n REFs preceding amode) to yield an object which possesses any of the modes REF^{n} amode, REF^{n-1} amode, REF^{n-2} amode, ..., REF amode, amode. The actual mode of the object yielded in an ALGOL 68 program is determined by the syntactic position of the object.(For a more detailed explanation see coercion.)

ALGOL 68 procedures require that the modes of each of the parameters (if any) and also the mode of the value yielded by the procedure be specified for every procedure declaration. A procedure which accepts an integral value as its first parameter and a real variable as its second parameter and returns a value of mode boolean would be represented by: PROC (INT, REF REAL) BOOL. This representation names a new mode; a value of this new mode is an appropriate routine denotation. Since PROC (INT, REF REAL) BOOL is a new mode,it may be used as a building block in the creation of other modes (i.e., [] PROC (INT, REF REAL) BOOL, REF PROC (INT, REF REAL) BOOL, etc.).

A variable declared to be of a united mode (using the union notion) may contain at any time a value of one of the constituent modes of the union. For example, a variable declared with the mode REF UNION (INT, REAL) may possess a value of mode INT or of mode REAL (only one at any particular time). Language facilities are provided to allow the programmer to test a variable of a united mode to see which mode it possesses at any particular time and to extract its value. Note: there are no values of a mode which begin

with UNION; all values assigned and retrieved from a united variable are values of one of its constituent modes.

In ALGOL 68 a structure is a mode. The following is a representation of a structured mode: STRUCT (REAL a, INT b,c). Unlike PL/I or COBOL the field names a, b, and c are part of the mode itself. To select a particular field from a structured variable such as STRUCT(REAL a, INT b,c)x, the programmer writes, for example, b of x. Assuming the above declaration for x, ALGOL 68 facilities allow the use of structured modes as aggregates as well as allowing for the selection of individual fields.

The Subset of Modes Chosen for Implementation

Prior to this implementation, the Oklahoma State University ALGOL 68 Compiler had a very limited mode capacity. Only variables of the modes REF BOOL, REF INT, REF REAL, REF CHAR, and REF COMPL (complex) and constants of mode PROC (procedure) were available. COMPL is not one of the basic modes of ALGOL 68; it is defined in the report to be of mode STRUCT (REAL re,im). However, COMPL has a full set of operators, so it does not hurt a programmer to think of COMPL as if it were a basic mode.

The design goal of this project was to increase the mode handling capacity of the compiler by a significant amount without adding the general list processing of modes which is required by a full mode implementation. The subset

selected adds the REF REF amode (pointers) and amode (constant) declaration facilities. Every mode (except procedured modes) of this subset can be represented by a descriptor of fixed size (see Chapter III). Procedured modes require linked lists to retain the modes of each parameter and the mode of the value yielded by the procedure.

All modes which may legally be declared in the subset must develop to a mode of the form REF^i [ ]^j REF^k BASIC_MODE or REF^i [ ]^j REF^k PROC_MODE.(REF^i, for example, means that there are i occurrences of the symbol REF with i being any integral value such that $i \geq 0$.) Figure 1 provides a formal grammar in modified Backus-Naur Form (17) of the subset of modes allowed in this implementation.

Coercion

Coercion is the ALGOL 68 term for the automatic modification of an internal object during the elaboration of a program. Most higher level languages allow some form of data conversion to occur, such as, converting integral values to real values and vice versa (15) (18). There are five coercions allowed in the subset: deproceduring, dereferencing, widening, rowing and voiding.

It is often useful to create procedured modes which have no parameters. An example is the built-in procedure random; when random is invoked, a REAL value in the interval (0,1) is yielded. In order to invoke a procedure which has parameters in ALGOL 68, the programmer simply writes the

```
<MODE> ::= <LIST_OF_REFS>
           <LIST_OF_ROWS>
           <LIST_OF_REFS>
           <CHOICE_OF_BASIC_OR_PROC>
<LIST_OF_REFS> ::= <LIST_OF_REFS> REF |
                   .EMPTY**
<LIST_OF_ROWS>* ::= <LIST_OF_ROWS>[] |
                    .EMPTY
<CHOICE_OF_BASIC_OR_PROC> ::= <BASIC_MODE> |
                              <PROC_MODE>
<BASIC_MODE> ::= INT | REAL | COMPL | CHAR | BOOL | FORMAT
<PROC_MODE> ::= PROC <OPTIONAL_PARAMETER_LIST>
                <MOID>
<MOID> ::= <MODE>
           VOID
<OPTIONAL_PARAMETER_LIST> ::= ( <MODE_LIST_PROPER> ) |
                              .EMPTY
<MODE_LIST_PROPER> ::= <MODE_LIST_PROPER><MODE> |
                       <MODE>
```

* "[] []" may be abbreviated "[,]",
  "[] [] []" may be abbreviated "[,,]", etc.

**.EMPTY represents the empty string.

Figure 1. Formal Grammar for Modes Allowed in the Subset

identifier symbol (or a unit which returns an object of mode procedure) followed by the actual parameters (i.e., A(2.0,"the string")). The appearance of an object of mode PROC REAL (no parameters) in the elaboration of the program does not always require its invocation. For example, if the assignation a:=b occurs where a is mode REF PROC REAL and b is mode PROC REAL, then the object which is to be assigned is the routine (b) which is specified by the right hand side of the assignation, not the value yielded by a call to that routine. The proper processing of procedured modes which have no parameters is the function of the deproceduring coercion. When the coercion deprocedure is applied to a mode, the resulting action is to invoke the procedure being coerced.

The dereferencing coercion causes an object of the mode REF amode to become an object of mode amode. That is, it causes the object to be modified, to possess the value to which it refers. For example, an object of the mode REF REF BOOL is a pointer which refers to a BOOL variable. If this mode is dereferenced, the result is the name (address) of the logical variable to which it refers. The mode of this new object is REF BOOL; if this mode is dereferenced, the result will be the value (TRUE or FALSE) which is possessed by the variable.

If a value of mode REAL is required and a value of mode INT is supplied, then in the proper syntactic positions the INT value will be widened to become a value of mode REAL.

Mode REAL may be widened to mode COMPL. Widening may only be applied to a value of the mode; it may not be applied to a variable. However, any syntactic position which allows widening also allows dereferencing. Thus, if a variable or pointer is provided, it will be dereferenced until a value is obtained.

Rowing allows a value of mode CHAR to become a value of mode [] CHAR or a name of mode REF CHAR to become a name of mode REF [] CHAR. This enables a programmer to use a scalar value (or name) in some positions where a multiple valued object is required. A prime example of this is where an object of mode CHAR is to be assigned to a variable of mode REF STRING (mode STRING is equivalent to mode [] CHAR). The CHAR value is rowed to become mode [1:1] CHAR; then the assignation may take place. This coercion is required because the denotation "A" is a mode CHAR value and it is often necessary or desirable to assign a CHAR of this type to a STRING variable. Another use of rowing is in parameter passing where a procedure of mode PROC ([] INT) REAL is provided a scalar INT value as an actual parameter. The INT value will be rowed and the resulting "multiple" value will be supplied to the procedure.

Voiding is used when the object yielded by some piece of code (such as, a routine or unitary clause) is to be discarded. There are three cases:

1) The object is of mode REF^{i} PROC moid (moid = amode or VOID) and the name was not yielded by a

confrontation (assignation, identity relation, or cast).

2) The object is of mode REF^{i} PROC moid; however, its value was yielded by a confrontation.

3) The object is not of mode REF^{i} PROC moid.

In case 1 the object is dereferenced i times, then deprocedured; the resulting value from the routine invocation is then voided. In cases 2 and 3 the mode is simply changed to VOID and any value is discarded. Figure 2 shows examples of the 3 cases and describes the actions to be taken.

Determining the Proper Coercion Sequence

There are three things which uniquely determine the coercion sequence to be applied to a value of some mode:

1) The a priori mode of the available object (coerend).

2) The a posteriori mode of the object required by the coercion (coercee).

3) The syntactic position (or sort) of the object.

There are five sorts of syntactic position, they are: strong, firm, meek, weak, and soft. Figure 3 shows, for each sort the valid coercions which may be applied and also some of the language constructs which give rise to each sort. Figures 4 through 7 are state diagrams which show the valid coercions allowed in the subset. In order to determine if it is possible to coerce mode A to mode B given a particular syntactic position, it is necessary to select

Case 1--Mode REF^{i} PROC moid (not yielded by confrontation).

```
PROC x:= REAL: y:= 2.0 * y;

x;
```

The mode of x is REF PROC REAL, because of the last ";" (occurring in the 2nd line), the name is to be voided.

ACTIONS
- Step 1--Dereference to mode PROC REAL.
- Step 2--Deprocedure - The routine "REAL: y:= 2.0 * y" is now invoked.
- Step 3--The real value yielded is discarded.

Case 2--Mode REF^{i} PROC moid (yielded by a confrontation).

```
PROC REAL x;        # x is of mode REF PROC REAL #

x:= REAL: y:= 2.0 * y;
```

ACTIONS
- Step 1--The value of mode PROC REAL is assigned to the variable x.
- Step 2--Step 1 yields an object of mode REF PROC REAL discard the result.

Case 3--Object not of mode REF^{i} PROC moid

```
REF REAL x;          # x is of mode REF REF REAL #

REAL y;              # y is of mode REF REAL #

x:= y;
```

ACTIONS
- Step 1--Assign the name y to the pointer x.
- Step 2--Void the pointer of mode REF REF REAL.

Figure 2. Example of the Three Cases Which Arise in Voiding

the appropriate diagram and beginning in state 1 follow the available arcs modifying the mode according to the label of the arc followed.

Sort	Coercions	Constructs
Strong	Deprocedure Dereference Row Widen Void	Actual parameter, the enclosed clause of a cast, the right hand side of an assignation, statements.
Firm	Deprocedure Dereference	Operands in a formula
Meek	Deprocedure Dereference	Units in FROM, BY and TO clauses, and trim-scripts
Weak	Deprocedure Dereference	Primary of a slice
Soft	Deprocedure	Left hand side of an assignation, one side of an identity relation

Figure 3. Coercions Allowed for Each Syntactic Position

For example, assume we are to coerce the mode REF PROC REF REAL to the mode REAL and further assume we are in a firm position. First, we select the graph of Figure 6 (sort is Firm), the coercion sequence is as follows:

REF PROC REF REAL	(a priori mode)
PROC REF REAL	M1 to M2 (dereference)
REF REAL	M2 to M3 (deprocedure)
REAL	M3 to M2 (dereference)

It is therefore possible to coerce mode REF PROC REF REAL to mode REAL in a firm position.

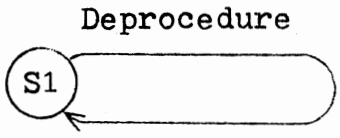
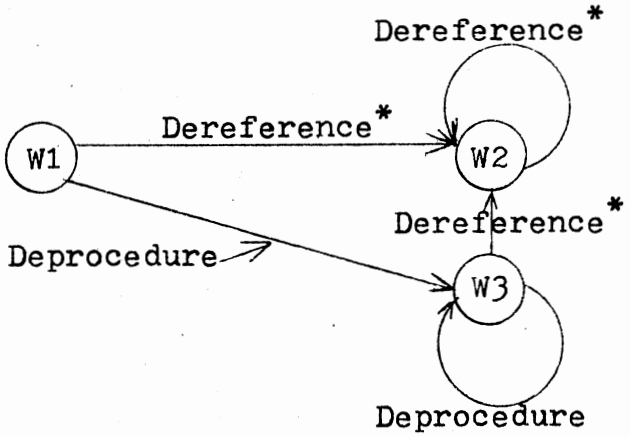


Figure 4. Soft State Diagram



Dereference* - means amode may not be coerced from REF amode

Figure 5. Weak State Diagram

Note: that in many states of the diagrams there is more than one possible arc to traverse from any node. This ambiguity can always be resolved by examination of the a priori and a posteriori modes (the coercion algorithm is shown in Appendix B).

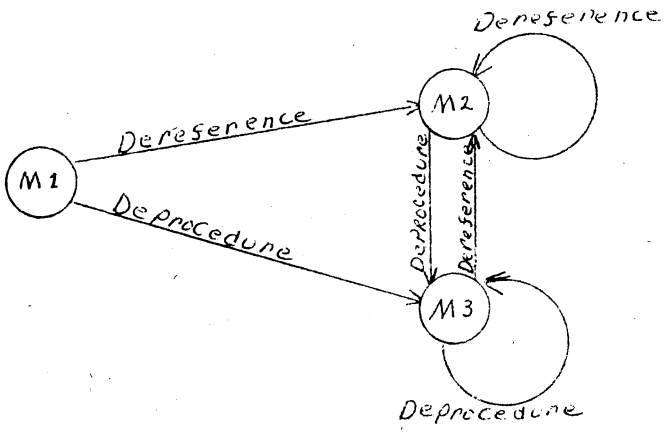


Figure 6. Meek and Firm State Diagram

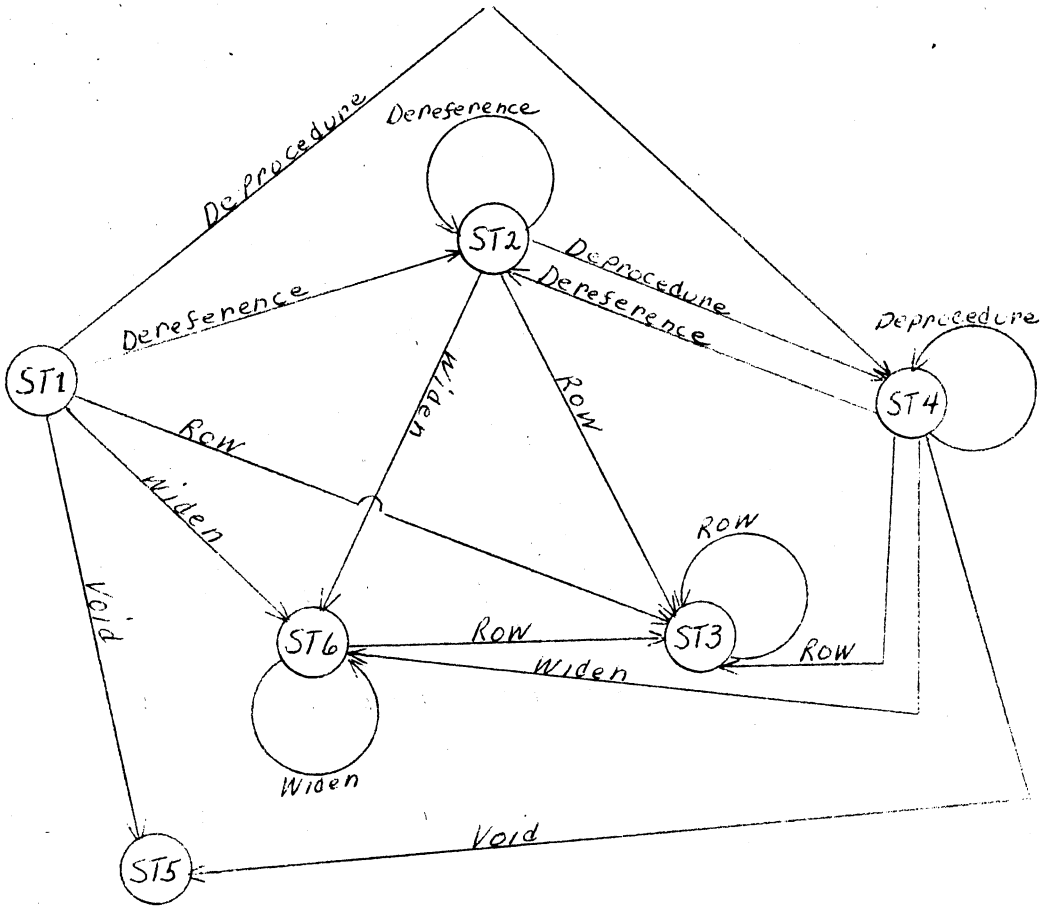


Figure 7. Strong State Diagram

Balancing

Conditional clauses, case clauses, and serial clauses which have multiple completion points (EXIT symbols) provide the capability of yielding values from different parts of the clause. For example, the conditional clause

IF p THEN x + 1.0 ELSE 4.0 FI

can return a value of x + 1.0 if p is TRUE or a value of 4.0 if p is FALSE. It is required that all alternative yields of a clause be of the same mode. Balancing provides the automatic mechanism for determining the mode of the yield of such a clause. In order to achieve a balance, it is necessary to know the mode of each of the alternative yields of the clause. The syntactic position of the clause is also required since the coercions applied to at least one of the alternatives must be only those coercions allowed upon a coercend in the same syntactic position as the clause. All other alternative yields in the clause are assumed to be in a strong position.

Appendix B contains a balancing algorithm for modes of the subset. The essential method of the balancing algorithm is to compute a target mode m = REF^{i} $\lceil\_\rceil^{j}$ REF^{k} BASIC_MODE. Where i, j, and k are the maximum values of the corresponding fields in all of the alternative yielding modes. The computed value of BASIC_MODE is the widest mode of each of the constituent BASIC_MODES with the BASIC_MODES arranged in the following order (widest last): PROC, BOOL, FORMAT, CHAR, INT, REAL, and COMPL. After the target mode is

computed the algorithm attempts to coerce all of the alternative modes to the target mode, recording the greatest coercion strength required. If all alternatives can be coerced to the target mode and the smallest strength coercion is less than or equal to the coercions allowed for the syntactic position of the clause, then the clause is balanced. If failure was due to the strength of the required syntactic position, then the clause cannot be balanced. If failure was due to failure of one or more of the modes of the alternatives to be coerced to the target mode, a new target mode is computed by applying either dereferencing or deproceduring to the target mode and the process is restarted. It should be noted that due to the manner in which the target mode was selected there is only one possible coercion which can be applied to it. If no coercion can be performed upon the target mode the balance fails.

This description of the balancing algorithm is oversimplified and does not account for the correct treatment of procedured modes. The balancing algorithm has been implemented in PL/I. Appendix C contains some examples of balancing using the PL/I implementation.

CHAPTER III

SYMBOL TABLE STRUCTURE

Original Structure

The most restrictive data structure of the previous versions of the ALGOL 68 compiler at Oklahoma State University was the symbol table. In order to reduce processing time the decision was made to retain the symbol table in internal memory (except for superceeded entries). Since the original version was implemented on an 8K machine, only a minimum amount of information about a variable could be maintained.

The logical structure of the symbol table consisted of three parts: the active symbol table, inactive symbol table and the block nesting table. Each unique non-keyword symbol was assigned a value for its internal identifier. The values assigned were integral values which began at minus one and decreased by one for each new symbol. The number which has been assigned to a symbol becomes its internal identifier and the key to the symbol table. A symbol table entry consisted of two words: word one contained the mode of the variable and word two contained the block for which the identifier was declared. The mode was encoded in the word and was of the form 10 * R + M where R is the number of rows

in a rowed mode and M is the basic mode of the variable. Figure 8 contains the allowable basic mode codes used in the previous version of the compiler.

Mode Number	Mode	Internal Coded Symbol
1	INT	409
2	REAL	411
3	COMPL	401
4	BOOL	405
5	CHAR	406
6	STRING*	404
7	LABEL	
8	PROC	410

*Not used

Figure 8. Allowable Modes for Previous Versions

When a declaration for a symbol was encountered in a new block a symbol table entry which was made for a previous instance of that symbol needed to be saved. For example, given the following segment of code:

```
(1)  BEGIN
(2)       INT a
            .
            .
            .
(3)       BEGIN
```

```
(4)             REAL a
                  .
                  .
                  .
(5)         END
(6)  END
```

after line 4 has been parsed a new declaration for the symbol a was indicated. Provisions for saving superseded declarations in new blocks and restoring the old symbol table entries upon block exit were made in the overflow symbol table. The overflow symbol table was physically located on a file (the simulated program memory file on disk). As a new declaration was encountered the old symbol table entry was saved. The format of the overflow symbol table entry was: current block number, identifier number, mode of the old declaration, and block in which the old declaration was made.

The block nesting table was created prior to the recognition of declarations and was physically located at the end of the active symbol table area in main memory. Each block in the program was numbered according to the position of its beginning symbol. The block nesting table consisted of the number of the block which immediately surrounded the current block as shown in Figure 9.

Revised Structure

It was necessary not only to expand the symbol table entries to retain more information, but the basic table maintenance method had to be revised, if a separate pass

to recognize declarations was to be made. With the previous symbol table structure when a block was exited all symbols which had been declared in that block were lost. A more permanent method was necessary in order to retain the information for code generation.

```
(1)  BEGIN                      Block    Containing Block
(2)      BEGIN                    1              0
(3)          BEGIN                2              1
                 END              3              2
(4)          BEGIN                4              2
                 END
             END
         END
```

Figure 9. Example of Block Nesting Table

The current symbol table comprises three parts: a tree of the source program structure, identifier lists and mode table. The symbol table is physically located on disk which is accessed through a software implemented paged memory system. The program structure tree is a binary tree which is built during the declaration recognition phase. This data structure replaces the block nesting table and represents the various ranges included in the ALGOL 68 program. The node used for the tree structure is shown in Figure 10.

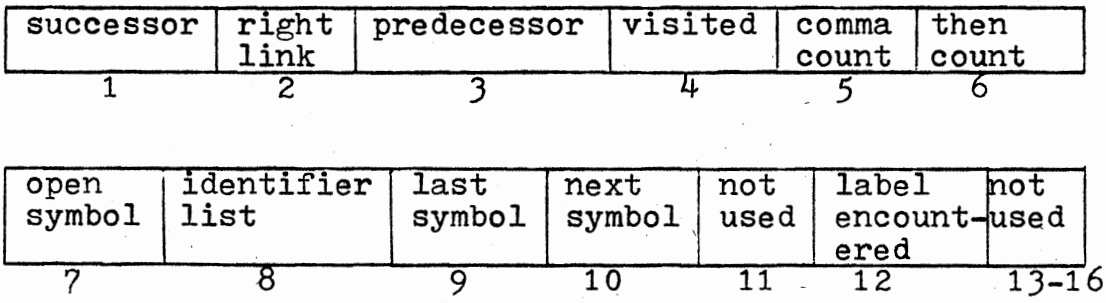


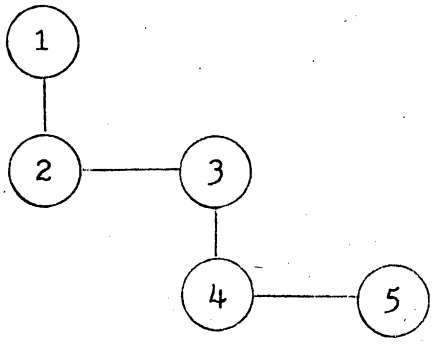
Figure 10. Tree Structure Node

A description of the uses of each field in the tree structure node follows. The successor, right link and predecessor fields are used to maintain the binary tree structure. Figure 11 shows the binary tree structure generated for an example program, along with a representation of what the tree pointer fields would contain. Notice in Figure 11 that two blocks which are on the same level (such as 2 and 3) are connected by right link pointers. When the nesting level increases, a successor pointer is used. The predecessor pointer provides the same information as the old block nesting table provided previously.

The visited field is used to provide an easy method for traversing the tree structure after it has been built and in the cross reference listing phase. The field is initialized to zero and is increased by one when the current node has been processed.

The fields named: comma count, then count, and open symbol type have the purpose of determining the type of enclosed clause this symbol table node represents.

```
(1)  BEGIN
(2)       BEGIN
          END
(3)       IF
(4)            THEN
(5)            ELSE
          FI
     END
```



Node	Successor	Right Link	Predecessor
1	2	-	-
2	-	3	1
3	4	-	1
4	-	5	3
5	-	-	3

Figure 11. Example Program Structure

Figure 12 provides a table which shows how the data from these fields are combined to determine the clause type.

The identifier list field points to the head of the identifier list associated with this block. If the field is null (a minus one) then there are no declarations in this range and block entry or block exit instructions are not generated.

The last symbol and next symbol fields are used in combination to determine in which syntactic position

the enclosed clause appears. Figure 13 shows some of the combinations which are used to signal the various positions.

Comma Count	Then Count	Open Symbol	Type Clause
not used	0	WHILE	LOOP CLAUSE
not used	0	DO	LOOP CLAUSE
0	0	(	SERIAL CLAUSE
0	>0	(	CONDITIONAL CLAUSE
>0	0	(	COLLATERAL CLAUSE
>0	>0	(	CASE CLAUSE
≥0	not used	⌜	TRIMSCRIPT

Figure 12. Determining Type of an Enclosed Clause

Last Symbol	Next Symbol	Sort
; or (	;	VOID
; or (	:=	SOFT
; or (	OPERATOR	FIRM
OPERATOR	; or)	FIRM
:=	; or)	STRONG
: (up to symbol)	⌟ or ,	MEEK
(	)	(same as the context of the surrounding range)

Figure 13. Determining the Context of an Enclosed Clause

The label encountered field is initialized to zero; when a label is inserted in the identifier list this value is set to a one. After the label encountered field is a one, it is not possible for a user declared symbol to be placed in the symbol table for the current range, although a temporary variable may be inserted in the table at any time.

Identifier List Nodes and the Mode Table

Figure 14 shows the fields used in the identifier list nodes. One node is created for each identifier declared in a range.

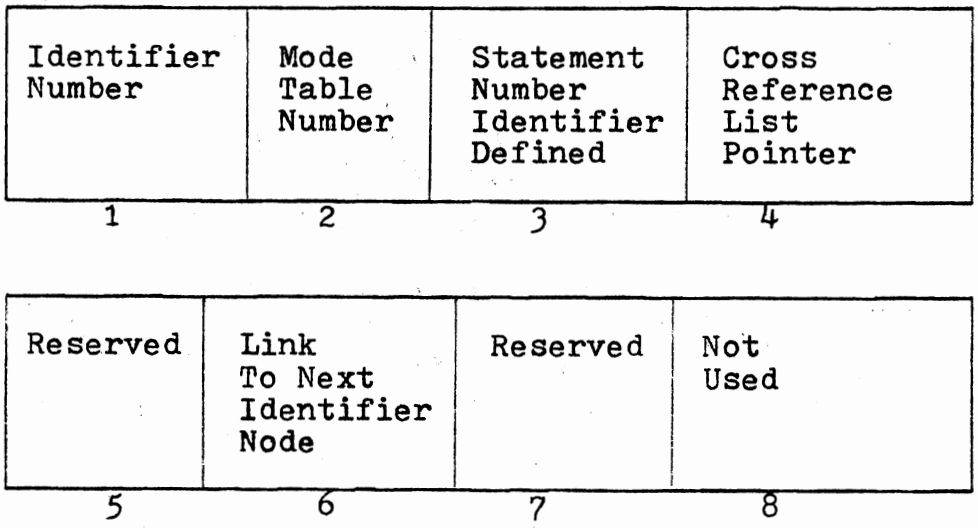


Figure 14. Identifier List Node

A description of each of the fields comprising the identifier node follows. The identifier number field is assigned during the lexical analysis of the source program. The identifier list nodes are maintained in descending sequence of the identifier number.

The mode table number is the index of the mode table entry which defines the mode for this identifier. The statement number and cross reference list pointer fields provide data for a cross reference listing which the programmer may specify as an optional output of the compilation. The two reserved words (5 and 7) are for the later addition of operator declarations to the compiler.

The mode table is physically located in the symbol table disk file. The mode table is assigned contiguous storage in order to allow fast access. Several standard modes are loaded into the table to provide compatability with existing mode references. The mode table entry for a mode of the form $REF^i \lceil \rfloor^j REF^k$ BASIC_MODE is shown in Figure 15. The mode table entries for modes of the form $REF^i \lceil \rfloor^j REF^k$ PROC (MODES) moid are fiven in Figure 16.

REFs before rows (=i)	Number of rows (=j)	REFs after rows (=k)	Negative of the BASIC_MODE number (see Figure 8)

Figure 15. Mode Table Entry for Modes of the Form $REF^i \lceil \rfloor^j REF^k$ BASIC_MODE

REFs before rows (=i)	Number of rows (=j)	REFs after rows (=k)	-8	Linked list of mode numbers of each parameter	Mode number of the yielding mode
1	2	3	4	5	6

Figure 16. Mode Table Entry for Modes of the Form
$REF^1 \lceil\rfloor^j REF^k$ PROC (MODES) moid

Declarations of the form MODE.A = REF B (where B is a user defined mode or a basic mode) are allowed, the general form for a mode of this type is $REF^i \lceil\rfloor^j REF^k$ MODE_NAME. Figure 17 displays the mode table entry for this type of mode declaration.

REFs before rows (=i)	Number of rows (=j)	REFs after rows (=k)	Negative of basic mode number or a pointer to another mode table entry	Linked list of node numbers of each parameter	Mode number of yielding mode	*
1	2	3	4	5	6	7

*Pointer to the list of symbols which constitute the actual row bounds

Figure 17. Mode Table Entry for Modes of the Form
$REF^1 \lceil\rfloor^j REF^k$ MODE_NAME

CHAPTER IV

FEATURES OF THE REVISED OKLAHOMA STATE UNIVERSITY ALGOL 68 COMPILER

Introduction

This chapter presents a description of the new and revised features of the ALGOL 68 compiler implemented as a part of this thesis. The new features are concerned generally with the extended mode handling capacity which was described in Chapters II and III. The original Oklahoma State University ALGOL 68 Compiler was based upon the definition in the original defining document of ALGOL 68 (4). The designers of the original definition felt a need to revise it slightly after the Oklahoma State University implementation effort had begun. Since it was necessary to add a new pass to the compiler (see Chapter V) in order to recognize properly declarations, the syntax recognized by the revised version is that of the revised report (5). This chapter also describes those features which have been modified to conform to the revised report. Appendix D contains a description of the grammar recognized by the revised version of the Oklahoma State University ALGOL 68 Compiler. When a capitalized word appears in this thesis surrounded by "<" and ">" (such as <UNIT>), it refers to a meta-symbol

in the grammar described in Appendix D.

Changes to Declarations and Modes

The previous versions of the compiler required that all declarations in an <ENCLOSED CLAUSE> preceed any other <UNIT> s in the clause. This restriction was the result of a size limitation imposed upon the original implementation. With the addition of a new pass to recognize declarations, this restriction is now removed. The only restriction imposed upon mixing <UNIT> s and declarations in an <ENCLOSED CLAUSE> is the ALGOL 68 language restriction that all declarations must preceed the first label in a <SERIAL CLAUSE>.

Previous versions of the compiler severly restricted the types of initialization expressions and row-bounds expressions which could be used in declarations. These expressions were limited to denotations, simple variables and simple variables preceeded by a monadic plus or a monadic minus operator. The current version allows full unitary clause (<UNIT>) facilities to be used both in row-declarers and in initialization of variables.

A limited form of mode declarations has been implemented for this thesis. A mode declaration allows the programmer to define a symbol to represent a user defined mode. The programmer may then use the symbol to stand for the newly defined mode in declarations, casts, and routine texts. Figure 18 provides some examples of valid mode declarations.

Declaration	Defined Mode
MODE .A = [i] INT	[] INT
MODE .B = REF .A	REF [] INT
MODE .C = CHAR;	CHAR
MODE .D = [i] .C	[] CHAR
MODE .E = [i] .D	[,] CHAR

Figure 18. Some Valid Mode Declarations

A symbol which is used to represent a mode in a mode declaration must be a stropped identifier. A stropped identifier is a standard identifier immediately preceeded by one of the stropping characters (. or '). The symbol .A is distinct from the symbol A and the two symbols may not be used interchangeably.

There are three important implementation restrictions upon mode declarations: no mode declaration may contain its own mode indication (symbol which stands for the mode), the developed mode must be a legal mode as defined for this implementation (see Chapter II), and no mode indication may be used before it is defined. Examples of mode declarations violating the first restriction are:

MODE .A = REF .A and

MODE .B = PROC (.B) REAL.

This restriction is consistant with full ALGOL 68 when the mode indication is not shielded within a struct or a union (this is not yet implemented).

An example of a mode declaration violating the second restriction is:

```
MODE .A = REF [i] INT,
     .B = [j] .A.
```

The developed mode for this example would be [j] REF [i] INT, but this implementation does not allow any symbols to occur between two row-of symbols ("[]").

The third restriction is violated by a mode declaration of the follwing type:

```
MODE .X = REF .Y,
     .Y = REF INT.
```

The effect of this particular declaration may be achieved by simply reversing the order of the symbol declarations

```
MODE .Y = REF INT,
     .X = REF .Y.
```

This restriction is necessary because the mode definitions are not recognized prior to the declarations being parsed.

The previous versions of the compiler made no distinction between identity declarations and variable declarations. This allowed (INT a = 3; a:= 2) to be accepted as a valid program. The revised version will correctly identify the assignation a:= 2 to be in error.

Procedure declarations which utilize procedure variables, row-of-procedure variables and constants may be made. In addition, the pre-existing facility of procedure constants remains available. Figure 19 displays an example of how a procedured variable may be declared, assigned

routines, and invoked.

```
(1)  BEGIN
(2)       PROC (REAL) REAL trig;
(3)       INT sw;
(4)       REAL nbr;
(5)       read ((sw,nbr));
(6)       trig := IF sw = 0 THEN sin ELSE cos FI;
(7)       print (trig(nbr))
(8)  END
```

Figure 19. Use of a Procedure Variable

Line 2 declares the identifier "trig" to be of mode REF PROC (REAL) REAL. Trig is a variable capable of possessing a routine. The function of the routine is not defined; however, any routine which is assigned to trig must have one REAL formal parameter and it must return a REAL result. Lines 3, 4, and 5 declare two variables (sw and nbr) and input values for those variables from the Standard Input file (STANDIN). Line 6 is an assignation, the right hand side of this assignation is a conditional clause. If the value of sw is zero then the routine sin (mode PROC (REAL) REAL) will be yielded by the clause; the routine cos (mode PROC (REAL) REAL) will be yielded, otherwise. Since the modes of both alternative yields are identical,

the clause will return a value of mode PROC (REAL) REAL which is either the routine sin or cos depending upon the value of sw. The yield of the clause is compatible with the left hand side of the assignation (the variable trig) so that the routine yielded by the conditional clause is assigned to the procedure variable trig. Line 7 causes the procedure variable trig to be invoked with an actual parameter value of nbr. The action taken (which routine is elaborated) depends upon which routine was assigned to trig in line 6.

Row-of procedure constants and variables were included to maintain the orthogonality of the ALGOL 68 language. That is, given that amode is a valid mode then [] amode is also a valid mode (this is not necessarily true in this subset). A value of a [] amode mode consists of a vector of amode values. PROC (REAL, INT) BOOL is a valid mode; therefore, a programmer might desire to declare an object of mode [] PROC (REAL, INT) BOOL or REF [] PROC (REAL, INT) BOOL. The value of a row-of-procedure object consists of a vector of routines, each assigned to an element of the row-of-procedure constant or variable. Figure 20 provides an example of a situation where a row-of-procedure constant is used. Line 2 declares the identifier "func" to be of the mode [] PROC (REAL,REAL) REAL. The virtual parameters are required on all declarers of row-of-procedure constants and variables because there is no <ROUTINE TEXT> for row-of-procedure modes. Lines 3 through 6 comprise a special

clause called a collateral clause. The collateral clause is being used here as a row display. The collateral clause is discussed below. At this time, the assumption that the collateral clause yields a vector of routines which are assigned to the constant func is sufficient. Line 12 causes the jth routine in the row-of-procedure variable to be invoked, passing as actual parameters the **REAL** values possessed by the variables a and b.

```
(1)  BEGIN
(2)       /_\ PROC (REAL, REAL) REAL func =
(3)       ((REAL x,y) REAL : x + y,
(4)       (REAL x,y) REAL : x - y,
(5)       (REAL x,y) REAL : x * y,
(6)       (REAL x,y) REAL : x / y);
(7)       INT j,
(8)       REAL a,b,c;
(9)       WHILE read (j);
(10)            j > 0 and j < 5
(11)      DO read ((a,b));
(12)         c:= func /_j_\ (a,b);
(13)         print ((j,a,b,c))
(14)      OD
(15) END
```

Figure 20. A Program Example Using a Row-of-Procedure Constant

Collateral, Conditional and Loop Clauses

A collateral clause is a clause which returns a value for each of the comma separated <UNIT> s in the clause. The value returned is treated as a value of a rowed mode (row display). The mode of the row display is normally determined by the balance of the clause; however, due to implementation restraints the balance mode of a collateral clause is assumed to be the same as the mode of its first unit. ALGOL 68 permits collateral clauses to be used in strong positions. Figure 21 shows some of the valid uses of collateral clauses.

Initializing a [ ] INT variable

```
[5] INT a:= (1,2,3,4,5);
```

The right hand side of an assignation

```
a:= (2 * a [1], 3 * a [2], a [3] + a [4],0,0);
```

As an actual parameter of a call

```
PROC sum = ([ ] REAL x) REAL:
     (REAL tot:= 0;
     FOR i TO upb x DO tot + := x [i] OD;
     tot);
     print (sum (1.2, 2.3, 3.4))
          #value printed = 3.9#
```

Figure 21. Allowable Uses of the Collateral Clause

Futher restrictions which are placed upon row displays are: they must not be used as actual parameters in calls to transput routines (print, put, putf, etc.), nor can they be nested to obtain row-row mode values.

The syntax for a <CONDITIONAL CLAUSE> has been changed in this implementation to allow a <SERIAL CLAUSE> in positions where only a list of unitary clauses was previously permitted. As an example, the following conditional clause would now be valid:

```
IF p THEN INT a; read (a); a ELSE 2.0 FI.
```

There are several ranges defined within conditional clauses, they are: between the IF and the FI, the THEN clause, the ELSE clause, and between an ELIF and its corresponding FI. Figure 22 displays the ranges of two example conditional clauses.

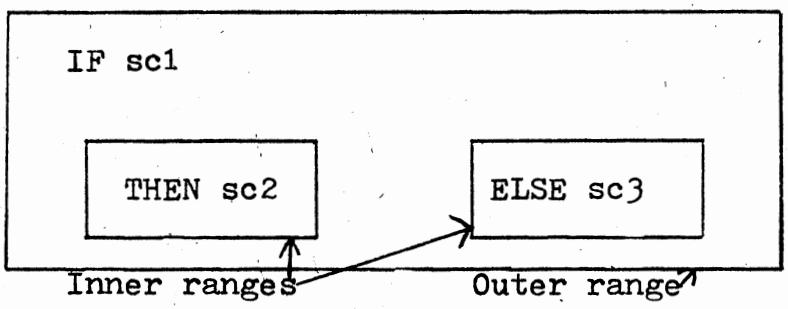
The changes made to the loop clause structure represent changes which make the revised version and the previous versions of the compiler incompatible. The syntax according to the original report allowed a single <UNIT> as the object of the loop clause (4). The revised report introduced the symbol OD to match the symbol DO and allows a <SERIAL CLAUSE> as the object of the loop (5). New syntax allows the following loop clause:

```
TO 5 DO REAL a; read (a); sum +:= a OD
```

The previous version would have required the following statements to achieve the same result:

```
TO 5 DO BEGIN
        REAL a;
        read (a);
        sum + := a
    END;
```

```
IF sc1 THEN sc2 ELSE sc3 FI
```




```
IF sc1 THEN sc2
       ELIF sc3
            THEN sc4
            ELSE sc5
FI
```

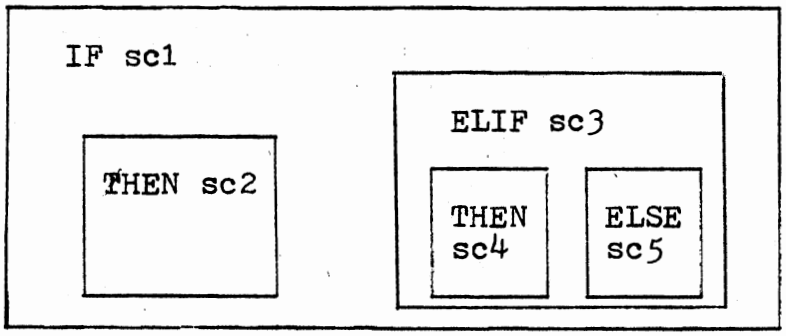


Figure 22. Ranges of Two Conditional Clauses

Along with the new syntax is a new definition of ranges in the loop clause. Figure 23 illustrates this new range definition.

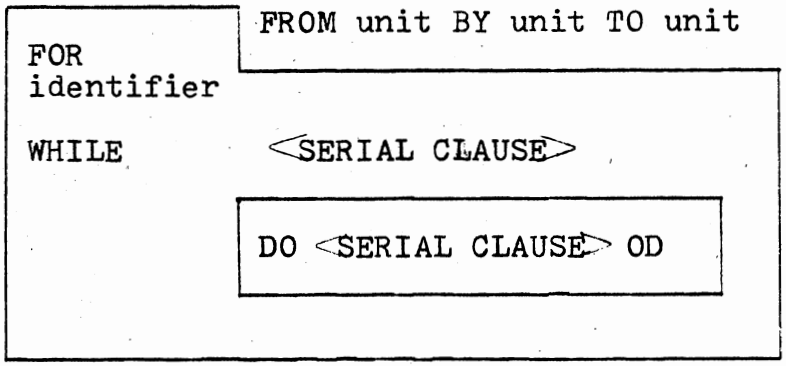


Figure 23. Ranges in the Loop Clause

The identifier defined in the FOR part can be accessed only in the WHILE and DO parts. Any declarations made in the WHILE part may be referenced in the DO part. The syntax accepted by the new version will also allow a loop clause to be the outermost range in the program i.e., the program DO SKIP OD is correct; however, the lexical analyzer will not accept this since it does not treat loop clauses as ranges. Due to time constraints this was not changed.

Identity Relations and Casts

Identity relations allow the testing of two REF amode variables to see if they refer to the same location (have the same name). There are two identity relation

operators: the IS relator (:=:) and the IS NOT relator (:¬=:). Given the declarations

REAL x := 3.14, y := 3.14;

the identity relation x :=: y yields false because the addresses (names) of x and y are different even though the values contained in those addresses are the same. It is not possible to use an identity relation between variables of two different modes (after balancing). Figure 24 is a sample program which displays the features of identity relators.

```
BEGIN
    REF INT a, INT b := 3, c := 3;
    a := b;  #'a' now refers to the location of 'b'#
    a :=: b;  #yields true#
    a :¬=: b;  #yields false#
    print (a);  #prints the value 3#
    a :=: c;  #yields false#
    a :¬=: c;  #yields true#
    a = c  #yields true#
END
```

Figure 24. Some Examples Using Identity Relators

A cast allows the programmer to change the strength of the position of an enclosed clause. The enclosed clause of

a cast is a strong position; therefore, any legal coercion may be applied to the value yielded by the enclosed clause. A cast is created by a mode indication followed by an enclosed clause such as: REAL (1 + 2). The enclosed clause will return an integral result which will then be widened to a real value (regardless of the available syntactic strength).

CHAPTER V

PLANNING FOR THE IMPLEMENTATION

Introduction

The planning for this implementation comprised three steps: 1) overall familiarization with the existing compiler, 2) designing the mode facility to be implemented, and 3) devising the syntactic analysis needed to recognize the program block structure.

Familiarity with the existing compiler was obtained by examining the theses of Jensen (1), Berry (2), and Eyler (3) and also by examination of the compiler code.

Modes

Planning for mode implementation required two major decisions, they were: selecting the subset of modes to be allowed and designing algorithms to perform the required mode manipulations and designing the symbol and mode tables. An objective of this thesis is to introduce an enhancement in the mode handling facilities for the Oklahoma State University ALGOL 68 Compiler. This was to be done without requiring a major rewrite of the code generation and interpretive execution phases of the compiler. The existing code relies heavily upon the codes used for the modes (see

Figure 8); therefore, any changes made had to preserve these numbers. This was attained by using the position of the defining mode entry in the mode table as the mode symbol and entering the modes listed in Figure 8 into the first eight locations in the mode table. With this convention a real mode, for example, still is represented by the number 2.

The mode subset was selected to allow REF amode entries as the major enhancement. This feature along with the orthogonalization of modes is a suitable beginning to the task of adding a full mode handling facility to the compiler.

After designing the subset of modes the compiler would accept, algorithms which would perform the functions of: coercion, balancing, and determining modes in assignations were devised. These algorithms were implemented in PL/I, and tested to insure that they were acceptable. The algorithms coded in ALGOL 68 can be found in Appendix B and test results of the PL/I implementation can be found in Appendix C.

The symbol table structure was chosen because this same type of symbol table was implemented for a class project. It is versatile enough to handle the block structuring of ALGOL 68. The symbol table structure also figures heavily into some recommendations for future enhancement of the compiler (see Chapter VII).

Syntactic Analysis

In order to recognize declarations in a pass prior to code generation, it was necessary to perform enough syntactic analysis to determine the blocking structure of the program. Several attempts where made at devising a grammar for the language, that would also be acceptable to the SLR(1) table generator developed by Joseph Gray (19). After substituting some "terminal symbols" for some syntactic entities which were not in fact terminals (and invoking other parsing algorithms to recognize these "terminal symbols"), it was possible to generate a grammar which would perform the required analysis. The resulting parser was; however, too large to be used practically, given the size restrictions imposed upon the compiler. It would have been possible to have used sparse matrix techniques to reduce the size of the parsing tables from 15,000 words ($\sim$150 states by 100 symbols) to about 3400 words but considering the size of the semantic routines, the author felt it was impractical to implement on a computer with 8K words of memory.

The syntactic analysis used is essentially a hand coded push down automaton which is similar to the methods used in the other phases of the compiler.

CHAPTER VI

IMPLEMENTATION

General Structure

In this chapter the modifications which have been made to the compiler are discussed. The compiler is a four pass compiler with an interpretive execution phase (Phase 5). Figures 25-30 are diagrams of the flow of data through the phases of the compiler. The flow of control for the compiler is Phases 1, 2, 3, 3.5, and 4. Execution is accomplished by Phase 5 which may be directly invoked (using actual pseudo machine code) or executed after compilation.

Phase 1 reads the :JOB card, performs analysis of the options selected by the programmer, prints the compiler options and sets various flag fields to be used by later phases of the compilation.

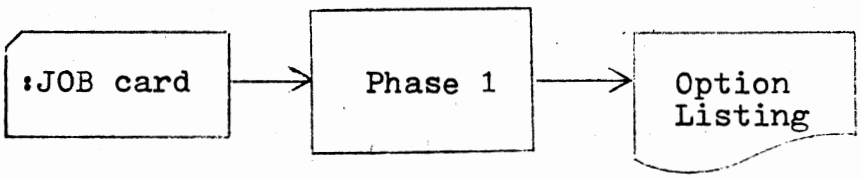


Figure 25. Phase 1-Job Card Analysis

Phase 2 performs a lexical analysis of the ALGOL 68 source program. Output consists of one integer per symbol in the source program. The only key words recognized as reserved symbols at this point are BEGIN, END, IF, FI, CASE, and ESAC. A source program listing is printed if it was requested on the job card. Any denotations encountered during the lexical analysis are converted to internal form and stored into the simulated program memory for Phase 5. A table of all symbols which were encountered in the lexical analysis is also passed to Phase 3 in common storage.

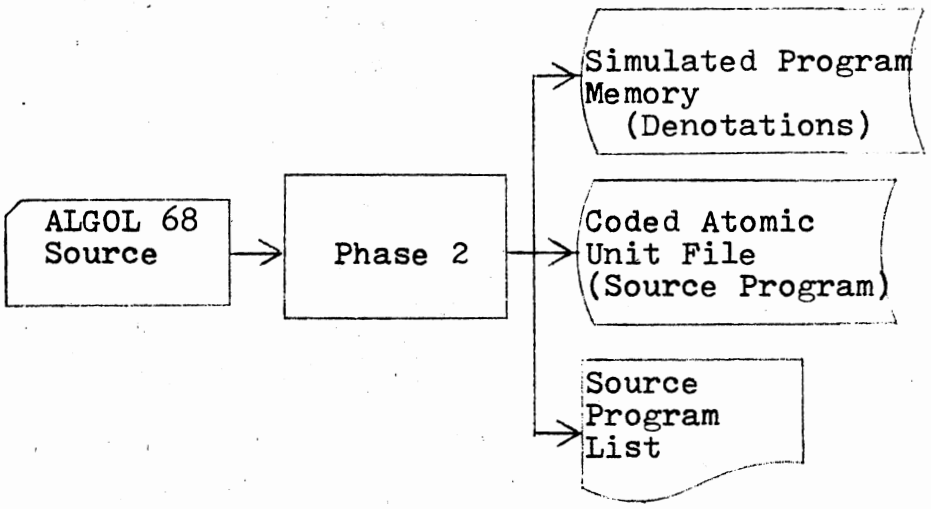


Figure 26. Phase 2-Lexical Analysis

Phase 3 performs two functions: first the table of symbols (received from Phase 2) is examined and all key-word symbols are identified, next a pass is made through the

source code file updating the keyword symbol numbers to reflect their special values (see Jensen (1)). The identifier table is also compressed (removing keyword symbols) and the corresponding changes are made to the source code.

The second part of Phase 3 changes a colon symbol which is preceded by a mode declaration to another code, to indicate that a routine follows. The left parenthesis of a formal parameter pack (if one exists) also is changed to a special symbol at this time. As a final function Phase 3 writes the variable name symbols to a disk file for debug output purposes.

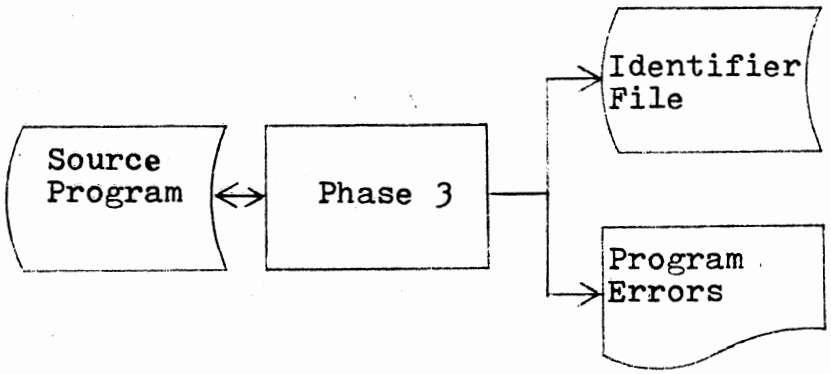


Figure 27. Phase 3-Keyword Recognition

Phase 3.5 is an entirely new pass written for this project. It has the primary function of recognizing declarations and building the symbol table entries for these declarations. Phase 3.5 also analyzes the blocking structure of the program in order to build the symbol table tree

structure. The output code file has all declaration symbols deleted. This phase modifies some other symbols in order to make Phase 4 parsing easier (such as using separate symbols for each different meaning of the colon symbol). Some object text is included as a part of the source text. This object text is never seen by the main section of Phase 4, but instead is placed immediately in simulated program memory by the Phase 4 input routine.

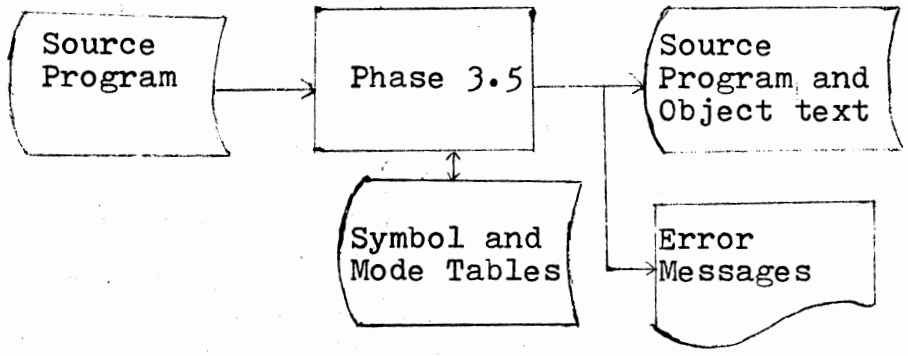


Figure 28. Phase 3.5-Declaration Recognition

Phase 4 is the main syntactic recognition and code generation phase of the compiler. The input consists of the modified source text and symbol table output from Phase 3.5. Output from Phase 4 consists of the generated object code and any applicable error messages.

The code generated by Phases 3.5 and 4 consists of instructions defined for a pseudo machine. Phase 5 performs the simulation of the generated pseudo machine code.

Phase 5 may also be executed as a stand alone program which executes object code loaded from cards.

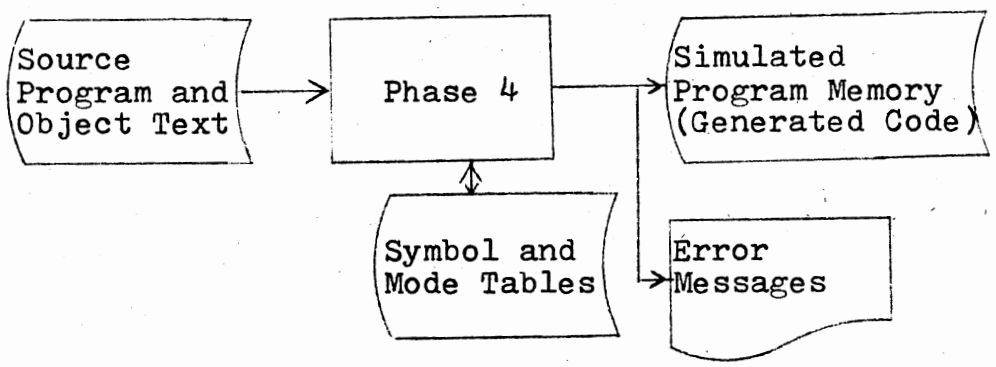


Figure 29. Phase 4-Code Generation

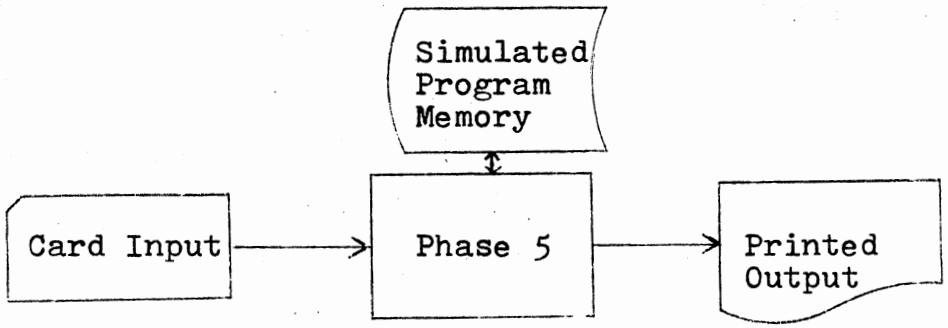


Figure 30. Phase 5-Interpretive Execution

Modifications Made to Phases 2 and 3

Phase 2

No modifications were made to Phase 1. Phase 2

modifications were concerned with stropping symbols. The symbols for stropping (' and .) immediately preceeding an identifier symbol cause that symbol to be treated differently from the same identifier symbol which is not stropped. ".ABC" is not the same as "ABC"; however, ".ABC" is equivalent to "!ABC". Stropped identifiers may be used as mode indicants. Figure 31 is the finite state automaton used to recognize stropped symbols.

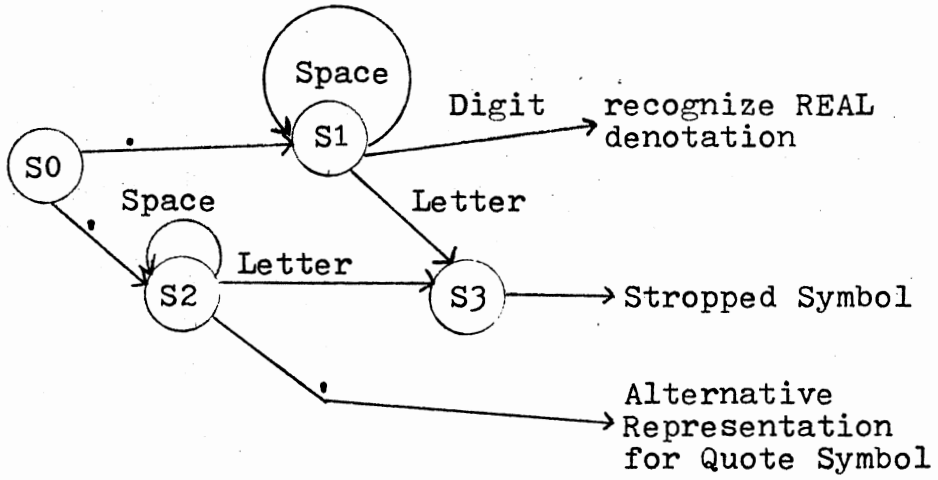


Figure 31. Recognition of Stropped Symbols

When a symbol is determined to be a stropped symbol the Code 617 is placed in the code file preceeding the identifier number for the symbol.

Phase 3

Phase 3 modifications consisted of minor modifications

to the keyword identification routine to identify correctly the additional symbols required to implement mode declarations and modifications required by the revised report (5). The additional symbols and their codes are:

Symbol	Code
OD	616
MODE	623

The pass to update the source code after the keywords have been recognized was rewritten completely in order to support five functional modifications:

1) The keyword OD is recognized as the symbol which terminates a loop clause.
2) The stropping symbol is removed from the source text if it precedes a keyword (.IF is equivalent to IF).
3) If the colon symbol is immediately preceded by a mode indication then the colon symbol is a routine symbol. The source text is then scanned from right to left to find the opening parenthesis of the formal parameters pack (if one exists). After the open symbol has been found it is replaced by a special code (Code=47) so Phase 3.5 will be able to immediately recognize a routine denotation. The previous version of the compiler required routine denotations to occur only in PROC constant declarations so this type distinction was not necessary (all routines were preceded by the symbol PROC).

4) Previous versions of the compiler used this pass to identify labels, which were then output on a special label file. Entries in the label file indicated the block number of each label declared in the program. Phase 4 needs this information at the beginning of a block in order to generate the allocate symbol instruction upon block entry.

   The revised version performs label identification in Phase 3.5. Here labels are entered in the symbol table along with other declarations. Phase 4 then searches the identifier list associated with a block and generates the allocate symbol instructions.

5) Phase 3 was also used to recognize the block structure of the program in order to create the block nesting table. The need for this table has been eliminated with the inclusion of the tree structured symbol table.

Phase 3.5

This phase consists of four major sections of Code:

Determine Nesting Level of the ALGOL 68 Source Program

A hand coded push down automaton recognizes the nesting level of the program which is reflected in the tree associated with the symbol table. This section analyzes

special symbols to determine whether they are loop clause symbols or declaration symbols. If a symbol is in one of these two categories then the appropriate subprogram to parse the construct is called. If the symbol is related to the nesting structure of the program (except for loop clauses) it handles the processing directly.

If the current symbol is a symbol which terminates a unit (such as ; , | etc.) then the status of the parse is examined and control returns to the location in the analysis which was interrupted due to the need to recognize a unit. If a symbol does not fall into one of the previous classes it is simply copied onto the output file.

Symbol and Mode Table Manipulation

Subprograms are included which allow for the manipulation of the symbol table. These subprogram functions comprise:

1) increasing the nesting level of the tree structure (build the tree structure),
2) decreasing the nesting level of the tree structure,
3) inserting an identifier into the symbol table,
4) searching the symbol table for the occurance of an identifier, and
5) allowing access to the simulated virtual storage which contains the symbol table.

In addition to the subroutines to manipulate the symbol table several routines are included which manipulate the

mode table. Given the number associated with a mode it is possible to fetch the mode table entry or given the mode table entry it is possible to obtain the corresponding mode number.

Special processing is required for the insertion of a mode entry into the mode table. Not only must the mode entry being processed be added to the table, but any modes which can be derived from that mode by the standard coercions or by slicing, also must be added. The insertion routine automatically derives these related modes and inserts them into the mode table. Given the mode REF REF [,] REF INT, Figure 32 displays all of the related modes which must be inserted into the mode table.

Mode Number	Mode	Comment
1	REF REF [,] REF INT	original mode
2	REF [,] REF INT	dereference mode 1
3	REF [] REF INT	slice mode 2
4	[,] REF INT	dereference mode 2
5	[] REF INT	slice mode 4
6	REF INT	subscript mode 5
7	INT	dereference mode 6

Figure 32. Modes Derived From REF REF [,] REF INT by Coercion and Slicing

Figure 33 is a flowchart which presents the algorithm required for the insertion of derived modes.

Loop Clause Processing

This section of code provides all processing necessary to recognize the nesting level associated with loop clauses. The loop clause recognizer also makes the necessary symbol table entries for the index of the FOR loop as well as any labels encountered in the serial clauses of the WHILE and DO . . . OD parts of the loop clause.

Declaration Processing

Declaration processing accounts for over 50% of the code of Phase 3.5. A large part of the complexity involved in the processing of declarations is a result of the recursive nature of the language. It is possible for declaration processing to be suspended in order to recognize a unitary clause (which may of course contain other declarations), and then be resumed after the unitary clause has been recognized. This facility requires mutually recursive co-routines which tend to obscure the clarity of FORTRAN subprograms. An example of a situation where this occurs is: REF (/ (INT I; READ (I); I) /) INT J. Upon encountering the first left parenthesis, recognition of the first declaration (declaration of J) is suspended and partial results saved (in the symbol table area). Flags are set to indicate the state of the parse, then the routine to process

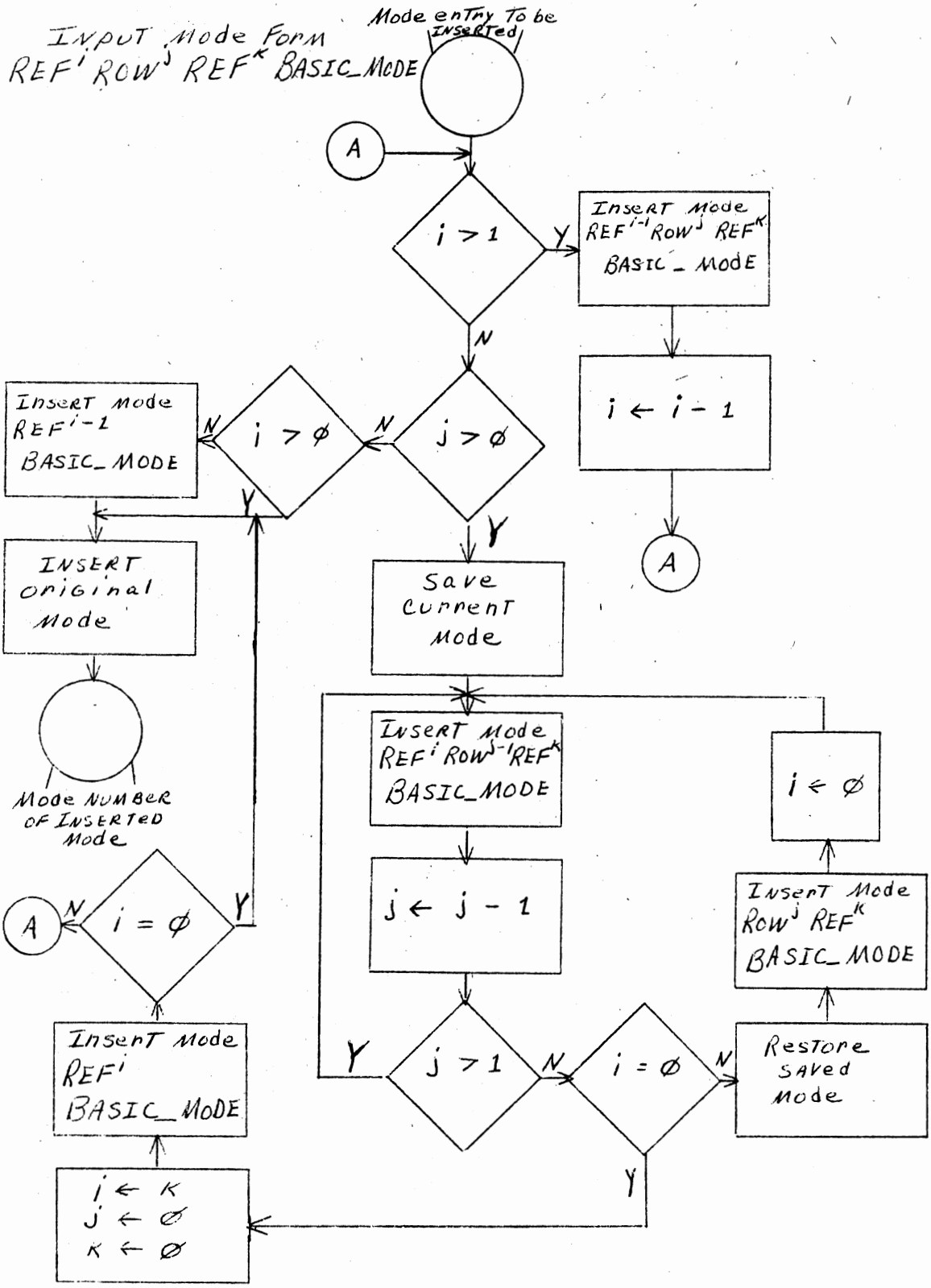


Figure 33. Related Mode Algorithm

program nesting structure is invoked. After the unit within the parenthesis has been recognized control returns to the location in the parse which was suspended.

The declaration parsing phase generates the code necessary for the allocation of variables during program execution. Code is also generated to update the statement number of any statements which have been deleted because they contained only declaration symbols.

The added feature of allowing any unit in array bounds declarations has forced modification to the previous methods of handling array allocation. A new source symbol (=49) has been introduced which, when encountered in Phase 4 causes the current unit being evaluated to be completed and the mode of the result is coerced meekly to mode INT. The lower and upper bounds of each row are left on the runtime stack. Figure 34 shows an example of the status of the runtime stack for the row declaration

⌜u1: u2, u3: u4, u5⌟ INT I.

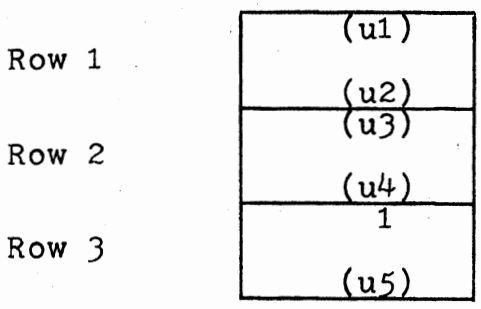


Figure 34. Status of the Runtime Stack During Elaboration of the Declaration ⌜u1: u2, u3: u4, u5⌟ INT I

The missing lower bound from the third row of the declaration of Figure 34 is assumed to be one. This requires a special instruction sequence to be generated whenever a comma or bus symbol (_/) follows the first unit of a row bounds pair. The special processing generates code to cause a constant of one to be loaded onto the runtime stack followed by a new pseudo machine instruction consisting of an operation code of a 67 (with all other fields set to zero). This new instruction will cause the top two integral values on the stack to be exchanged. Figure 35 provides an example of the output text generated from the array declaration given in Figure 34.

A mode declaration such as:

MODE .x = /_y_/ INT

poses some special processing problems during Phase 3.5. The units in the row declaration must be elaborated each time the mode indication occurs as a variable declaration. For example, given the above definition of the mode .x, Figure 36 shows two very different results depending upon the local declaration of y. The implementation of this is accomplished by saving the source code symbols of the mode declaration in a list which may be accessed through the mode table entry for the mode being defined. When the mode indication occurs as an actual variable declarer the reading of source text symbols switches from the source code file to the list associated with the mode entry. The current set of source text being accessed is determined from a stack.

u1	Source code for unit 1
49	End of unit 1
u2	Source code for unit 2
49	End of unit 2
u3	Source code for unit 3
49	End of unit 3
u4	Source code for unit 4
49	End of unit 4
u5	Source code for unit 5
49	End of unit 5
"301, 1, 1, 0"	Push a 1 onto the stack
"67, 0, 0, 0"	Exchange top two integral values
"501, -1, 3, xxx"	Allocate descriptor for array (xxx is the address of the skeleton descriptor).

Figure 35. Example of Output Text for the Example Array Declaration
/ u1: u2, u3: u4, u5 / INT I

When the stack is empty, source code is obtained from the input disk file. If the stack is not empty the stack contains pointers to the list associated with the mode being developed. A null symbol (-999) causes the stack top to be decreased and input resumes from the point at which it was last suspended.

```
BEGIN
     MODE .x = [y] INT;
     BEGIN
          INT y = 3;
          .x z;  #z is a vector of size 3#
          .x z1;  #z1 is also a vector of size 3#
          .
          .
          .
     END;
     BEGIN
          INT a:= 2;
          PROC y = INT: a * := 2;
          .x z;  #this z is a vector of size 4#
          .x z1;  #this z1 is a vector of size 8#
          .
          .
          .
     END
END
```

Figure 36. Two Different Results Using the Same Mode Definition

Phase 4

There are five major functions of Phase 4 which required significant modifications. Three of the five functions which were modified have been explained earlier in this thesis, they are:

1) Declaration processing was removed from Phase 4.

2) Symbol table access was provided to the new tree

structured symbol table.

3) The mode coercion and balancing algorithms were implemented in FORTRAN.

Mr. Eyler's (3) implementation of procedures, particularity with respect to parameter passing, was restricted due to the fact that no descriptor containing the modes of formal parameters could be maintained at compile time. A procedure call was executed by placing the actual parameters upon the runtime stack followed by an end of parameter flag. The routine contained retrieve parameter instructions which fetched the actual parameter, performed any required coercions and stored the value either in the symbol table or a local area depending upon the mode of the formal parameter. If the wrong number of parameters was passed or the mode of an actual parameter could not be coerced to the mode of the formal parameter the error was not detected until execution time.

The runtime symbol table uses one word to represent the actual mode of an object. This does not provide sufficient capacity to store all of the information about the expanded modes during execution. It was therefore necessary to modify the parameter passing mechanism to perform coercions upon parameters at the point of invocation. This is possible because the new compile time symbol table contains a descriptor of the modes of all formal parameters. It is therefore possible to announce at compile time when parameters have modes which do not match formal parameters, or

when the incorrect number of actual parameters is used.

The runtime mode descriptor does not allow for the indication of all of the modes allowed in the new subset. This is especially true for reference-to modes (pointers). The implemented solution has several drawbacks in the area of possible expansion to the runtime system. Reference-to variables are treated as integer modes by the runtime code. There is no confusion in handling the variables since the code generator does know the actual mode of the object and will not dereference a true integral value. The best solution would have been to revise the runtime mode descriptor to contain all of the required information.

Phase 5

Modifications to Phase 5 include implementing the code necessary to provide for the coercions which have been modified or added and modification of parameter passing mechanisms. New instructions include code to perform rowing and dereferencing.

When rowing is indicated, an array descriptor which has [1:1] in all row bounds is created. The address in the descriptor is set to point to the object being rowed. If the object being rowed is a variable, then the result is the address of the descriptor. If the object being rowed is a value, the descriptor itself is the result.

There are two different actions which can result from dereferencing. Given a mode of the form REF^{i} amode, when i

is greater than one, a dereference instruction yields an address of mode REF^{i-1} amode. If i is equal to one a dereference instruction yields an amode value.

Modifications in the parameter passing algorithm conform to the changes detailed in the discussion of Phase 4.

CHAPTER VII

SUMMARY, CONCLUSIONS AND FUTURE WORK

Summary

An implementation has been completed upon the IBM 360/65 which meets the criteria of limited portability and a significant expansion of the mode processing capabilities of the Oklahoma State University ALGOL 68 Compiler. The improvements made for this implementation include the following:

1) allows the use of full unitary clauses in declarations;
2) allows mixed unitary clauses and declarations in a range;
3) includes mode declarations for a subset of ALGOL 68 modes;
4) allows row displays to be used in a restricted context;
5) procedure variables have been implemented.

Conclusions

The mode processing capability of the Oklahoma State University ALGOL 68 Compiler has been enhanced significantly. A considerable amount of work will be required before

a full mode processing facility can be added to the Oklahoma State University ALGOL 68 Compiler.

Future Work

Implementation of United Modes

The mode table will handle the addition of united modes as shown in Figure 37.

Mode Number	REFs1	ROWs	REFs2	BASIC	Mode List
1	0	0	0	INT	0
2	0	0	0	REAL	0
3	0	0	0	CHAR	0
4	0	0	0	COMPL	0
5	0	0	0	UNION	1 →4
(6	0	0	0	UNION	1 →2→3→5) Unresolved
6	0	0	0	UNION	1→2→3→4 Resolved

Figure 37. Mode Table Entry for the United Mode UNION (INT, REAL, CHAR, UNION(COMPL, INT))

The mode list would be kept in numerical sequence; so when an attempt is made to add a mode to the list which matches a mode that is already on the list, it is not added. If one of the modes of a mode list for a mode table entry is a united mode the mode lists should be merged. Problems which

must be solved prior to successful implementation of united modes include: adding uniting to the coercion and balancing algorithms, and detection of related modes in a union. Two modes are related if they both can be coerced firmly from a common mode, such as PROC REF INT and REF INT; they both may be derived REF PROC PROC REF INT, for example.

Implementation of Structured Modes

The mode table representation of a structured mode would be very similar to that of a united mode. Figure 38 shows a possible method of managing structured modes.

Mode Number	REFs1	ROWs	REFs2	BASIC	Mode List	Field Selector List
1	1	0	0	INT	0	0
2	0	0	0	REAL	0	0
3	1	0	0	.A	0	0
4	0	0	0	STRUCT	2→3	x→y
5	0	0	0	STRUCT	1→1→4	a→b→c

Figure 38. Mode Table Entry for the Structured Mode MODE .A = STRUCT (REF INT a, b, STRUCT (REAL x, REF .A y) c)

In the case of united modes, when a mode list referred to another united mode, the mode lists are merged; however, the mode lists are not merged for structured modes. The structured mode entry contains a list of the field selector

names which correspond to a mode list entry. A potential problem which must be resolved prior to successful structure implementation is the identification of structure displays (how to distinguish it from a row display).

Source Program Representation by a Syntax Tree

Consider the symbol table tree structure representation for the example program segment given in Figure 39.

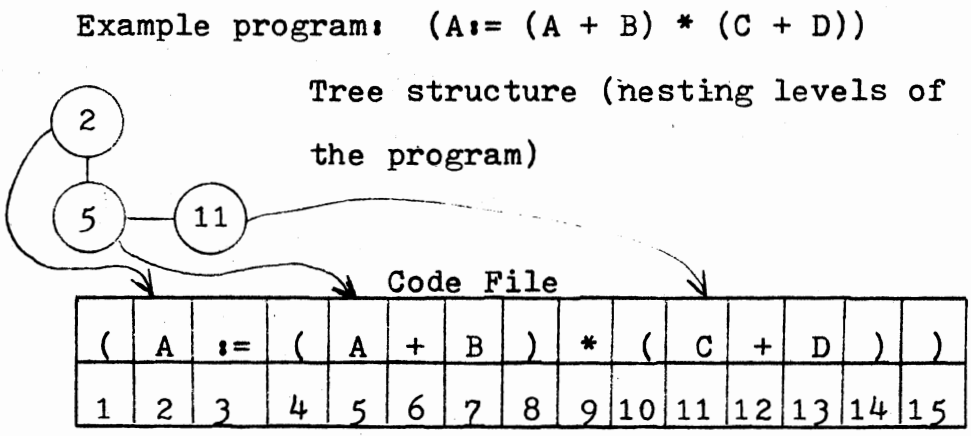


Figure 39. Tree Structure and Code File for the Simplified Example

The numbers within the tree structure nodes point to the position of the code file which contains the first symbol following the symbol which caused the tree structure node to be created. Using a $\downarrow$ symbol to represent a left parenthesis and a $\uparrow$ symbol to represent a right parenthesis, we append the code for a particular nesting level to the tree

structure nodes for that level. The results are shown in Figure 40.

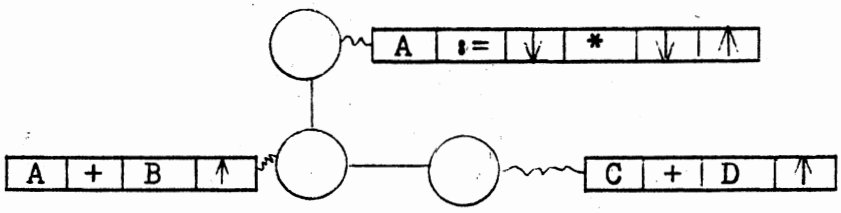


Figure 40. Tree Structure with Code Appended

A segment of code appended to a structure node may contain several ↓ symbols; however, a segment will contain exactly one ↑ symbol (which may be used to signal end of that particular code segment).

Figure 41 shows the tree after each individual code segment has been translated into prefix polish notation (preferable to postfix because it is easier to build a tree from). The ↓ symbols are treated as operands for the purpose of the polish string conversion.

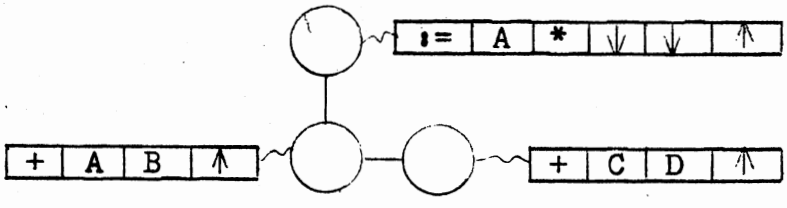


Figure 41. Tree Structure After Conversion of Code Segments to Prefix Polish

Figure 42 shows how the code segments can be translated into trees and interconnected with the tree structure nodes.

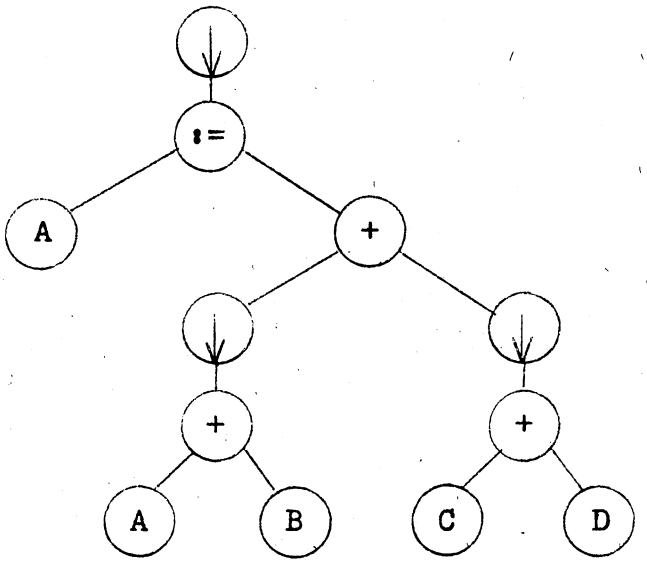


Figure 42. Syntax Tree for Sample Program Segment

After the entire program has been converted into a syntax tree, mode information can be appended to the various nodes (identifiers can be replaced by pointers to the symbol table). This structure would allow for coercion and balancing to be performed prior to actual code generation. Code generation becomes relatively simple after all coercions and balances have been added to the tree.

Implementation of unions and structures could be done in the time required to complete a masters thesis. Conversion of the source program to a syntax tree and code generation from that tree should be attempted only by

someone with sufficient time to complete the task.

String Implementation

The implementation of strings in the Oklahoma State University ALGOL 68 Compiler could be accomplished by several methods. One method would be to allocate a section of program storage for a string space. The method explained by David Gries (20) could then be used to manage strings. The amount of string storage space allocated could be controlled by an option in the :JOB card with an appropriate default value (say 2K words).

Another possible method of string implementation would be to use string descriptors as shown in Fiugre 43. With this method strings could be allocated on the stack in the same way as any other local variable. This would make possible automatic recovery of unused string space when a block which contains a string is exited. If a string expands beyond its boundary, a new segment of storage would be allocated and linked to the original segments.

New string segments may occur in storage areas reserved for blocks which are newer in scope than the original string segments. If this occurs, special treatment must be given to those segments at the time a block exit occurs from an inner block. These string segments must be moved from their previous location to the end of the storage area for the block which immediately surrounds the block being exited.

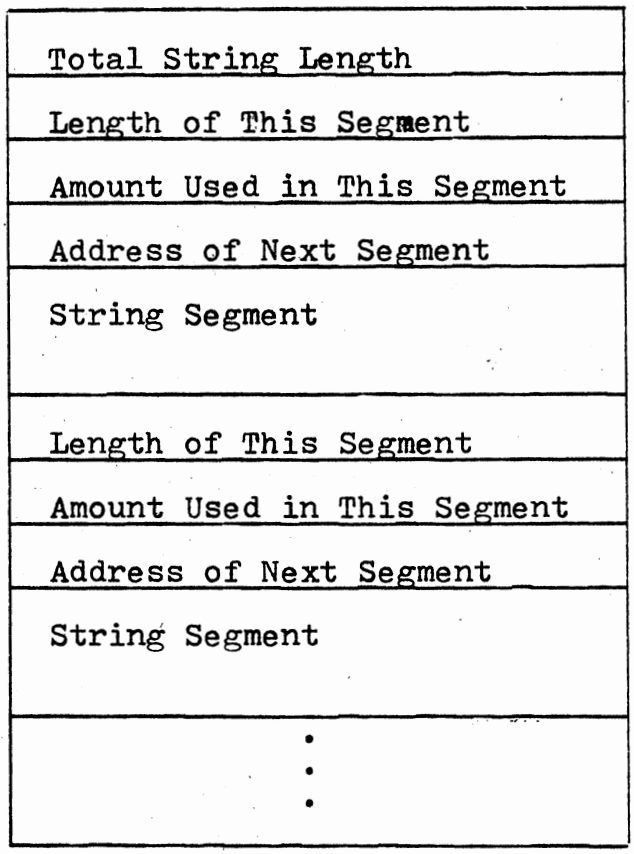


Figure 43. Possible String Descriptor Format

No matter which method is used for string implementation, new mode processing for mode STRING will be required. Mode STRING is equivalent to FLEX [1:0] CHAR. Mode equivalence implies that two objects of equivalent modes will have the same storage structure. [] CHAR is currently implemented very different from any reasonable method of string implementation; therefore, it will be necessary to introduce two new coercions, they are: string and unstring. These coercions would be valid in any strong or firm context

and would convert [ ] CHAR to STRING and STRING to [ ] CHAR respectively.

REFERENCES

(1) Jensen, J. C. "Implementation of a Scientific Subset of ALGOL 68." (Unpub. M.S. thesis, Oklahoma University, 1973.)

(2) Berry, R. "A Practical Implementation of Formatted Transput in ALGOL 68." (Unpub. M.S. thesis, Oklahoma State University, 1973.)

(3) Eyler, A. D. "The Implementation of a Subset of Procedures in an ALGOL 68 Compiler." (Unpub. M.S. thesis, Oklahoma State University, 1975.)

(4) van Wijngaarden, A. (Editor), B. J. Mailloux, J. E. L. Peck and C. H. A. Koster. "Report on the Algorithmic Language ALGOL 68." Numerische Mathematik, Vol. 14 (1969), pp. 79-218.

(5) van Wijngaarden, A. (Editor), B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisker. "Revised Report on the Algorithmic Language ALGOL 68." Supplement to ALGOL Bulletin No. 36. Vancouver: University of British Columbia, 1974.

(6) Peck, J. E. L. An ALGOL 68 Companion. Vancouver: University of British Columbia, 1971.

(7) Lindsey, C. H. and S. G. van der Meulen. Informal Introduction to ALGOL 68. Amsterdam: North Holland Publishing Company, 1973.

(8) Woodward, P. M. and S. G. Bond. ALGOL 68-R Users Guide. London: Her Majesty's Stationery Office, 1974.

(9) Currie, I. F., S. G. Bond and J. D. Morison. "ALGOL 68-R." ALGOL 68 Implementation. J. E. L. Peck (ed). Amsterdam: North Holland Publishing Co., 1971, pp. 21-34.

(10) Hedrick, G. E. (Editor). Proceedings of the 1975 International Conference on ALGOL 68. Stillwater: Oklahoma State University, 1975.

(11) Peck, J. E. L. "On Storage of Modes and Some Context Conditions." Proceedings Informal Conference on ALGOL 68 Implementation. Vancouver: University of British Columbia, 1969, pp. 70-79.

(12) Zosel, M. E. "A Formal Grammar for the Representation of Modes and its Application to ALGOL 68." (Unpub. Ph.D. dissertation, University of Washington, 1971.)

(13) Král, J. "The Equivalence of Modes and the Equivalence of Finite Automata." ALGOL Bulletin No. 35. Manchester: University of Manchester, 1973, pp. 34-35.

(14) Lane, H. J. "Coercion Methods Using Boolean Matrices." Proceedings Informal Conference on ALGOL 68 Implementation. San Francisco: University of San Francisco, 1973.

(15) IBM System/360 Operating System, PL/I (F), Language Reference Manual (GC28-8201-4).

(16) IBM OS Full American National Standard COBOL (GC28-6396-4).

(17) Backus, J. W. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceeding of the International Conference on Information Processing, UNESCO, Paris, 1959. Munich: R. Oldenbourg, 1960.

(18) IBM System/360 and System/370 FORTRAN IV Language (GC28-6515-10).

(19) Gray, J. L. "Implementation of a SLR(1) Parsing Algorithm." (Unpub. M.S. thesis, Oklahoma State University, 1973.)

(20) Gries, D. Computer Construction for Digital Computers. New York: John Wiley and Sons, Inc., 1971, pp. 180-181.

APPENDIXES

APPENDIX A

GLOSSARY OF ALGOL 68 TERMS

Alternative yields - Every clause yields a value. Some clauses have only one point from which a value can be yielded, others (such as conditional clauses) have two or more points in the code from which the value can be yielded. Each possible location from which a value can be yielded is termed an alternative yield.

Amode - Symbol used to stand for an arbitrary mode.

Assignation - Causing a variable of some mode to possess a value of the same mode.

Balancing - Clauses which yield values from more than one point must be balanced to insure that all of the yields are of a common mode.

Case clause - A simple case clause provides the capability to select a unitary clause to elaborate based upon an integral value. If the integral value is outside of the range of unitary clauses provided (i.e., less than 1 or greater than the number of unitary clauses provided) a serial clause (the OUT part) is elaborated. An extended case clause allows the nesting of case clauses in the OUT part (OUSE).

Cast - A cast allows a unit to be placed in a strong position (see Chapter II) and causes the value yielded by the unit to be coerced to the specified mode.

Coercee - The result after applying a coercion to a coercend is a coercee.

Coercend - The basic building blocks out of which units are constructed such as: assignations, formulas,

denotations and applied identifiers.

Coercion - Modifying the mode of a coercend to that required by its context with a corresponding modification to the value.

Conditional clause - A simple conditional clause provides the facility for making a true or false decision as to which program path will be elaborated (IF THEN ELSE FI). The extended conditional clause (ELIF) allows the nesting of conditional clauses in the ELSE path of elaboration.

Denotation - A construct strongly resembling a constant in other programming languages.

Deprocedure - This coercion causes a procedure which has no parameters to be invoked.

Dereference - This coercion removes one or more REFs from an object (yielding the value at which the REF amode object was pointing).

Develop - Mode declarations which contain mode indicants in their definitions must be developed by replacing the indicant by its corresponding definition.

Elaborate - The act of carrying out the actions defined by a program in a suitable environment.

Enclosed clause - A clause which is wholly contained between two bracketing symbols. Examples are CONDITIONAL CLAUSES, LOOP CLAUSES, etc.

Equivalencing - There are often many ways of defining the same mode. Mode equivalencing identifies the different

definitions to be the same mode.

Identity relation - An identity relation allows two names to be tested for equality or inequality.

Mode - Specifies the class to which a value belongs.

Name - The location or address of a value in the compiler.

Object - An object is either a value or a name (address) which refers to a value.

Orthogonality - The language design principle which requires that a given language construct should be allowed everywhere it is logically consistant.

Possess - An object is possessed by the symbol in the source program which causes it to exist.

Primary - A primary is a denotation, applied identifier or an enclosed clause.

Range - A range defines a segment of a program which contains local declarations. Any declarations found within a range may be accessed by other ranges contained within the original range. Objects declared within a range may not be accessed by any references which are contained in code which is external to the range.

Reference - An object of mode REF REF amode performs functions similar to the PL/I pointer variable. This object is said to reference the object at which it is pointing.

Routine denotation - The formal parameters, mode of the yield of the routine and the code comprising a routine.

Routine text - The code comprising a routine.

Rowing - This coercion allows a multiple value (or name) to be constructed from a scalar value (or name).

Serial clause - A serial clause is constructed from <UNIT> s and <DECLARATIONS>. The individual <UNIT> s and <DECLARATIONS> can be intermixed and must be separated by go on symbols (;).

Slice - A slice is an object which refers to a subset of a multiple value.

Sort - Same as syntactic position.

Syntactic position - The syntactic position of a coercend refers to the type of language construct in which the coercend appears. The syntactic position of a coercend determines the coercions which may be applied.

Trimscript - A subscript or a slice.

Virtual parameters - The mode of all parameters in a procedured mode must be specified in a declaration. The specification may be indirect by the formal parameter list of the routine denotation used to initialize a procedured constant or variable. In the absence of a routine denotation for procedure initialization the mode of each parameter must be specified by a virtual parameter pack. The virtual parameter pack follows the PROC symbol and consists of a list of the modes of each parameter (in order). The modes are separated by commas.

Voiding - This coercion causes a value yielded from some section of code to be discarded.

Widening - This coercion converts a value of mode INT to a value of mode REAL, also INT may be widened to COMPL, and REAL may be widened to COMPL.

Yield - The yield of a section of code is the value which that code makes available for further computation.

APPENDIX B

MODE PROCESSING ALGORITHMS

```
BEGIN

    SUBJECT: MODE PROCESSING ALGORITHMS FOR THE OSU ALGOL 68 SUBSET COMPILER

    AUTHOR: WALTER M. SEAY

    INSTALLATION: OKLAHOMA STATE UNIVERSITY

    DATE: SUMMER SEMESTER 1976

    PROJECT ADVISOR: DR. GEORGE HEDRICK

    /*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*

    SPECIAL NOTE: THESE ALGORITHMS HAVE BEEN HAND TRANSLATED FROM
                  A PL/I IMPLEMENTATION. THERE IS NO SUITABLE COMPILER
                  AVAILABLE AT OKLAHOMA STATE UNIVERSITY TO TEST THE
                  VALIDITY OF THE TRANSLATION.

    /*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*

    THIS PROGRAM ALLOWS FOR THE TESTING OF THREE MODE MANIPULATION
    ALGORITHMS, THEY ARE:
        1) COERCION
        2) BALANCING
        3) ASSIGNATION.

    INPUT CONSISTS OF AN ARBITRARY NUMBER OF INPUT SETS.  EACH INPUT
    SET CONSISTS OF A FUNCTION DEFINITION CARD FOLLOWED BY CARDS
    CONTAINING VARIABLE DATA DEPENDING UPON THE FUNCTION SELECTED.

    COERCION
    CARD 1      COLUMNS 1-6 = "COERCE"
    CARD 2      COLUMNS 1-80 = PUNCH THE A PRIORI MODE
    CARD 3      COLUMNS 1-80 = PUNCH THE A POSTERIORI MODE
    CARD 4      COLUMNS 1-6 = PUNCH THE STRENGTH OF THE SYNTACTIC POSITION

    BALANCING
    CARD 1    COLUMNS 1-7 = "BALANCE"
    CARD 2      COLUMNS 1-6 = PUNCH THE STRENGTH OF THE SYNTACTIC POSITION
                        8-9 = PUNCH THE NUMBER OF UNITS TO BE BALANCED
    CARD 3 TO 2 + (NUMBER OF UNITS TO BE BALANCED)
                COLUMNS 1-80 = PUNCH THE MODE OF ON UNIT

    ASSIGNATION
    CARD 1      COLUMNS 1-6 = "ASSIGN"
    CARD 2      COLUMNS 1-80 = DESTINATION MODE := SOURCE MODE

    VALID MODES WHICH MAY BE INPUT ARE DEFINED BY THE FOLLOWING
    BNF GRAMMAR.  UNDERLINED SYMBOLS IN THE GRAMMAR REPRESENTS NON TERMINAL
    SYMBOLS.  EMPTY REPRESENTS THE EMPTY STRING.

    VALID MODES ::= NON PROC | PROC MODE

    NON PROC    ::= REFS, ROWS, REFS, BASIC MODE

    PROC MODE   ::= REFS, ROWS, REFS, PROCS, NON PROC

    REFS        ::= REF, REFS | EMPTY
```

```
  ROWS          ::= (/ COMMAS /) | EMPTY

  COMMAS        ::= ",", COMMAS |  EMPTY

  BASIC MODE    ::= FILE | FORMAT | BYTES | CHAR | STRING |
                    BITS | BOOL | INT | REAL | COMPL

  PROCS         ::= PROC, PROCS | EMPTY

 VALID SYNTACTIC POSITION STRENGTHS (STRONGEST LAST)
  ARE:

      1) SOFT
      2) WEAK
      3) MEEK
      4) FIRM
      5) STRONG

#

#

  AMODE REPRESENTS THE CODED VERSION OF A MODE

#
MODE AMODE =  SIRUCT ( INT REFS1,
                           ROWS,
                           REFS2,
                      SIRING SIMPLEMODE,
                      INT NR_OF_PROC_LEVELS,
                           Y_REFS1
                           Y_ROWS,
                           Y_REFS2,
                      SIRING Y_SIMPLEMODE);
```

```
PROC PARSE = ( REF STRING STRING, REF AMODE MODE) VOID:

#

  THIS PROCEDURE ACCEPTS A STRING WHICH CONTAINS AN EXTERNAL
  REPRESENTATION OF A MODE AND PERFORMS A CONVERSION (USING AN FSA)
  TO INTERNAL FORM.

#
  BEGIN
    STRING SYMBOL,
    INT ACTION,ITEMP,STATE := 1,KEY,
    (/ , /) INT TABLE = ((8,7,2,4,0,0,3),
                         (0,0,0,0,0,0,0),
                         (8,7,0,4,0,0,3),
                         (0,0,0,0,5,4,0),
                         (8,7,0,0,0,0,6),
                         (8,7,0,0,0,0,6),
                         (0,0,0,0,0,0,0),
                         (8,7,2,10,0,0,9),
                         (0,7,0,10,0,0,9),
                         (0,0,0,0,11,10,0),
                         (0,7,0,0,0,0,12),
                         (0,7,0,0,0,0,12)),
                ATABLE = ((1,3,3,5,0,0,7),
                          (0,0,0,0,0,0,0),
                          (1,3,0,5,0,0,7),
                          (0,0,0,0,0,5,0),
                          (1,3,0,0,0,0,8),
                          (1,3,0,0,0,0,8),
                          (0,0,0,0,0,0,0),
                          (2,4,4,6,0,0,9),
                          (0,4,0,6,0,0,9),
                          (0,0,0,0,0,6,0),
                          (0,4,0,0,0,0,10),
                          (0,4,0,0,0,0,10));
    SIMPLEMODE OF MODE := Y_SIMPLEMODE OF MODE := "";
    REFS1 OF MODE := ROWS OF MODE := REFS2 OF MODE := 0;
    Y_REFS1 OF MODE := Y_ROWS OF MODE := Y_REFS2 OF MODE := 0;
    NR_OF_PROC_LEVELS OF MODE := 0;
      WHILE (SYMBOL := SCAN (STRING)) ¬= ""
        DO
          KEY := IF SYMBOL = "PROC"
                   THEN 1
                   ELIF SYMBOL = "VOID"
                     THEN 3
                     ELIF SYMBOL = "("
                       THEN 4
                       ELIF SYMBOL = ")"
                         THEN 5
                         ELIF SYMBOL = ","
                           THEN 6
                           ELIF SYMBOL = "REF"
                             THEN 7
                             ELSE 2
                 FI;
          ITEMP := TABLE (/STATE, KEY/);
          IF ITEMP = 0
            THEN PRINT ((NEWLINE, "PARSE ERROR"));
```

```
                STOP
        FI;
        ACTION := ATABLE (/STATE, KEY/);
        STATE := ITEMP;
        CASE ACTION
          IN
            #1#
            (SIMPLEMODE OF MODE := "PROC";
             NR_OF_PROC_LEVELS OF MODE := 1),
            #2#
            NR_OF_PROC_LEFELS OF MODE +:= 1,
            #3#
            SIMPLEMODE OF MODE := SYMBOL,
            #4#
            Y_SIMPLEMODE OF MODE := SYMBOL,
            #5#
            ROWS OF MODE +:= 1,
            #6#
            Y_ROWS OF MODE +:= 1,
            #7#
            REFS1 OF MODE +:= 1,
            #8#
            REFS2 OF MODE +:= 1,
            #9#
            Y_REFS1 OF MODE +:= 1,
            #10#
            Y_REFS2 OF MODE +:= 1
          OUT
            IF ACTION ¬= 0
              THEN PRINT ((NEWLINE, "PARSE ERROR"));
                   STOP
              FI
        ESAC
      OD;
      IF ¬(STATE = 2 OR STATE = 7)
        THEN PRINT ((NEWLINE, "PARSE ERROR"));
             STOP
      FI
END;
```

```
PROC SCAN = ( REF STRING STRING) STRING:

#

  THIS PROCEDURE RETURNS THE NEXT COMPLETE SYMBOL (NO IMBEDDED OR
  LEADING BLANKS) FOUND IN THE  STRING PARAMETER TO THE INVOKING
  PROCEDURE.

#
    BEGIN
      STRING SYMBOL := "", S2;
      IF STRING = ""
        THEN SYMBOL
        ELSE WHILE STRING(/1/) = ""
                DO
                  STRING := STRING (/2: /)
                OD;
                SYMBOL :=
                IF STRING (/1/) = "("
                  THEN STRING := STRING (/2: /);
                       "("
                  ELIF STRING (/1/) = ")"
                    THEN STRING := STRING (/2: /);
                         ")"
                    ELIF STRING (/1/) := ","
                      THEN STRING := STRING (/2: /);
                           ","
                      ELSE INT I = INDEX (STRING, " ");
                           S2 := STRING (/1:I/);
                           STRING := STRING (/I+1: /);
                           S2
                FI
    END;
```

```
PROC COERCE = ( REF AMODE FROM, TO,
                REF STRING SORT,
                REF INT LARGEST,
                REF BOOL MFLAG) VOID:

#

  THIS PROCEDURE COMPUTES THE UNIQUE COERCION PATH FROM MODE "FROM"
  TO MODE "TO" GIVEN THE STRENGTH OF THE SYNTACTIC POSITION "SORT"
  PROVIDED A VALID COERCION SEQUENCE EXISTS.

#
  BEGIN
    BOOL ERR,
    INT STATE := 1, FIRST := 0,
    (/5/) BOOL POS_VEC,
    (/ /) INT TO_STATE = (4,2,6,3,5);
    MFLAG := FALSE;
    LARGEST := 0;
    DO
      IF SIMPLEMODE OF FROM = "SKIP" AND SORT = "STRONG"
        THEN LARGEST := 5;
             FROM := TO;
             PRINT ((NEWLINE, "MODE SKIP COERCED TO "));
             PRINT_MODE (FROM);
             RETURN
        ELIF FROM = TO
          THEN PRINT ((NEWLINE, "MODES MATCH"));
               RETURN
          ELSE POSS (STATE, SORT, POS_VEC);
               PRINT ((NEWLINE, STATE, "  ", SORT, "  ", POS_VEC));
               SIFT (FROM, TO, POS_VEC, SORT);
               PRINT ((NEWLINE, STATE, "  ", SORT, "  ", POS_VEC));
               FOR I TO 5
                 DO
                   IF POS_VEC (/I/)
                     THEN IF FIRST ¬= 0
                            THEN PRINT ((NEWLINE,
                                         "MULTIPLE POSSIBILITIES"))
                                 LARGEST := -1;
                                 RETURN
                            ELSE FIRST := I
                          FI
                 OD;
                 IF FIRST = 0
                   THEN PRINT ((NEWLINE, "NO POSSIBLE COERCION"));
                        LARGEST := -1;
                        RETURN
                 FI;
                 LARGEST := (FIRST > LARGEST | FIRST | LARGEST);
                 CASE FIRST
                   IN
                     #1#
                     (DEPROCEDURE (FROM, ERR),
                      PRINT ((NEWLINE, "DEPROCEDURE")) ),
                     #2#
                     (DEREFERENCE (FROM, ERR);
                      IF REFS1 OF FROM = 0
                        THEN MFLAG :=  TRUE
```

```
                    EI;
                     PRINT ((NEWLINE, "DEREFERENCE")) ),
                    #3#
                    (WIDEN (FROM, ERR);
                     PRINT ((NEWLINE, "WIDEN")) ),
                    #4#
                    (ROW (FROM, ERR);
                     PRINT ((NEWLINE, "ROW")) ),
                    #5#
                    ( VOID (FROM, ERR);
                     PRINT ((NEWLINE, "VOID")) )
                ESAC;
                PRINT ((NEWLINE, "MODE AFTER COERCION"));
                PRINT_MODE (FROM);
                IF ERR
                  THEN PRINT ((NEWLINE, "COERCION ERROR"));
                       LARGEST := -1;
                       RETURN
                  ELIF SORT = "SOFT"
                    THEN STATE := 2;
                    ELIF SORT = "WEAK" OR SORT = "MEEK" OR
                         SORT = "FIRM"
                      THEN STATE := (FIRST = 1 | 3 | 2)
                      ELSE STATE := TO_STATE (/FIRST/)
                FI
    FI
  OD;
  RETURN: SKIP
END;
```

```
PROC PRINT_MODE = ( REF AMODE MODE) VOID:

#

  THIS PROCEDURE PRINTS THE EXTERNAL REPRESENTATION OF A MODE.

#
    BEGIN
      IO REFS1 OF MODE DO PRINT ("REF ") OD;
      IF ROWS OF MODE ¬= 0
        THEN PRINT ("(");
             IO ROWS OF MODE - 1 DO PRINT (",") OD;
             PRINT (") ")
      FI;
      IO REFS2 OF MODE DO PRINT ("REF ") OD;
      PRINT (SIMPLEMODE OF MODE);
      IF SIMPLEMODE OF MODE = "PROC"
        THEN IO NR_OF_PROC_LEVELS OF MODE - 1 DO PRINT (" PROC") OD;
             PRINT (" ");
             IO Y_REFS1 OF MODE DO PRINT ("REF ") OD;
             IF Y_ROWS OF MODE ¬= 0
               THEN PRINT ("(");
                    IO Y_ROWS OF MODE - 1 DO PRINT (",") OD;
                    PRINT (") ")
             FI;
             IO Y_REFS2 OF MODE DO PRINT ("REF ") OD;
             PRINT (Y_SIMPLEMODE OF MODE)
      FI
    END;
```

```
OP = = ( AMODE M1,M2) BOOL:

#

  THIS OPERATOR PERFORMS THE EQUAL COMPARISON FOR TWO OBJECTS OF
  MODE AMODE.

#
    IF REFS1 OF M1 ¬= REFS1 OF M2 OR
       ROWS OF M1 ¬= ROWS OF M2 OR
       REFS2 OF M1 ¬= REFS2 OF M2 OR
       SIMPLEMODE OF M1 ¬= SIMPLEMODE OF M2
       THEN FALSE
       ELIF SIMPLEMODE OF M1 ¬= "PROC"
            THEN TRUE
            ELIF Y_REFS1 OF M1 ¬= Y_REFS1 OF M2 OR
                 Y_ROWS OF M1 ¬= Y_ROWS OF M2 OR
                 Y_REFS2 OF M1 ¬= Y_REFS2 OF M2 OR
                 NR_OF_PROC_LEVELS OF M1 ¬= NR_OF_PROC_LEVELS OF M2 OR
                 Y_SIMPLEMODE OF M1 ¬= Y_SIMPLEMODE OF M2
                 THEN FALSE
                 ELSE TRUE
    FI;
```

```
PROC POSS = ( INT STATE, STRING SORT) (/ /) BOOL:

#

  THIS PROCEDURE RETURNS A VECTOR OF BOOLEAN VALUES WHICH INDICATES
  WHETHER OR NOT A PARTICULAR COERCION IS VALID IN THE GIVEN SYNTACTIC
  POSITION.

#
    BEGIN
      BOOL T = TRUE, F = FALSE;
      (/5/) BOOL POS_VEC := (F, F, F, F, F);
      IF SORT = "SOFT"
        THEN POS_VEC (/1/) := T
        ELIF SORT = "WEAK" OR
             SORT = "MEEK" OR
            SORT = "FIRM"
             THEN POS_VEC (/1:2/) := (T, T)
             ELIF SORT ¬= "STRONG"
                  THEN PRINT ((SORT, " IS INVALID"));
                       STOP
                  ELSE CASE STATE
                         IN
                           #1#
                           POS_VEC := (T, T, T, T, T),
                           #2#
                           POS_VEC := (T, T, T, T, F),
                           #3#
                           POS_VEC (/4/) := T,
                           #4#
                           POS_VEC := (T, T, T, T, F),
                           #5#
                           SKIP,
                           #6#
                           POS_VEC (/3:4/) := (T, T)
                       ESAC
      FI
  END;
```

```
PROC SIFT = ( REF AMODE IN_MODE, OT_MODE,
              [/ /] BOOL POS_VEC,
              STRING SORT) VOID:

#

THIS PROCEDURE EXAMINS THE A PRIORI MODE, THE A POSTERIORI MODE,
AND THE POSSIBILITY VECTOR AND SIFTS THE POSSIBILITIES UNTIL AT MOST
ONE COERCION EXISTS.

#
    BEGIN
      IF POS_VEC [/1/]
        THEN IF REFS1 OF IN_MODE ¬= 0 OR
                ROWS OF IN_MODE ¬= 0 OR
                REFS2 OF IN_MODE ¬= 0
              THEN POS_VEC [/1/] := FALSE
              ELSE IF SIMPLEMODE OF OT_MODE = "PROC" AND
                      NR_OF_PROC_LEVELS OF IN_MODE <=
                         NR_OF_PROC_LEVELS OF OT_MODE
                    THEN POS_VEC [/1/] := FALSE
                   FI
             FI
      FI;
      IF POS_VEC [/2/]
        THEN IF (SORT = "WEAK" | 1 | 0) >= REFS1 OF IN_MODE
               THEN POS_VEC [/2/] := FALSE
             FI
      FI;
      IF POS_VEC [/3/]
        THEN
          IF NOT (SIMPLEMODE OF IN_MODE = "INT" AND
             SIMPLEMODE OF OT_MODE = "REAL") OR
             (SIMPLEMODE OF IN_MODE = "INT" AND
             SIMPLEMODE OF OT_MODE = "COMPL") OR
             (SIMPLEMODE OF IN_MODE = "REAL" AND
             SIMPLEMODE OF OT_MODE = "COMPL")
            THEN POS_VEC [/3/] := FALSE
              ELIF ROWS OF OT_MODE = 0
                THEN POS_VEC [/3/] := FALSE
                ELIF NOT (SIMPLEMODE OF IN_MODE = "BITS" AND
                      SIMPLEMODE OF OT_MODE = "BOOL") OR
                      (SIMPLEMODE OF IN_MODE = "BYTES" AND
                      SIMPLEMODE OF OT_MODE = "CHAR)
                  THEN POS_VEC [/3/] := FALSE
          FI;
          IF ROWS OF IN_MODE ¬= 0 OR
             REFS1 OF IN_MODE ¬= 0 OR
             REFS2 OF IN_MODE ¬= 0
            THEN POS_VEC [/3/] := FALSE
          FI
      FI;
      IF POS_VEC [/4/]
        THEN IF ROWS OF IN_MODE ¬= 0 AND
                REFS1 OF IN_MODE ¬= 0 AND
                REFS1 OF OT_MODE ¬= 0
              THEN POS_VEC [/4/] := FALSE
             FI
             IF ROWS OF IN_MODE = 0 AND
```

```
            REFS1 OF IN_MODE ¬= REFS2 OF OT_MODE
          THEN POS_VEC (/4/) := FALSE
        FI;
        IF (SIMPLEMODE OF IN_MODE ¬=
            SIMPLEMODE OF OT_MODE) OR
            (ROWS OF IN_MODE >= ROWS OF OT_MODE)
          THEN POS_VEC (/4/) := FALSE
        FI;
        IF SIMPLEMODE OF IN_MODE = "PROC" AND
            (NR_OF_PROC_LEVELS OF IN_MODE ¬=
                NR_OF_PROC_LEVELS OF OT_MODE OR
            Y_REFS1 OF IN_MODE ¬= Y_REMODE OR
            Y_REFS2 OF IN_MODE ¬= Y_REFS2 OF OT_MODE OR
            Y_ROWS OF IN_MODE ¬= Y_ROWS OF OT_MODE OR
            Y_SIMPLEMODE OF IN_MODE ¬= Y_SIMPLEMODE OF OT_MODE)
          THEN POS_VEC (/4/) := FALSE
        FI
    FI;
    IF POS_VEC(/5/) AND SIMPLEMODE OF OT_MODE ¬= "VOID"
      THEN POS_VEC (/5/) := FALSE
    FI
END;
```

```
PROC DEPROCEDURE = ( REF AMODE MODE, REF BOOL ERROR) VOID:

#

  THIS PROCEDURE UPDATES THE GIVEN MODE TO REFLECT IT"S STATUS
  AFTER THE DEPROCEDUREING COERCION HAS BEEN APPLIED.

#
    BEGIN
      ERROR := IF REFS1 OF MODE +
                  REFS2 OF MODE +
                  ROWS  OF MODE ¬= 0
                THEN TRUE
                ELIF SIMPLEMODE OF MODE ¬= "PROC"
                  THEN TRUE
                  ELIF NR_OF_PROC_LEVELS OF MODE -:= 1 = 0
                    THEN (REFS1 OF MODE := Y_REFS1 OF MODE) := 0;
                         REFS2 OF MODE := Y_REFS2 OF MODE) := 0;
                         (ROWS OF MODE  := Y_ROWS   OF MODE) := 0;
                         (SIMPLEMODE OF MODE := Y_SIMPLEMODE OF MODE) := "";
                         FALSE
                FI
    END;
```

```
PROC DEREFERENCE = ( REF AMODE MODE, REF BOOL ERROR: VOID:

#

THIS PROCEDURE UPDATES THE GIVEN MODE TO REFLECT IT"S STATUS
AFTER THE DEREFERENCING COERCION HAS BEEN APPLIED.

#
 ERROR := IF REFS1 OF MODE < 1
            THEN TRUE
            ELSE REFS1 OF MODE -:= 1
          FI;
```

```
PROC WIDEN = ( REF AMODE MODE, REF BOOL ERROR) VOID:

#

  THIS PROCEDURE UPDATES THE GIVEN MODE TO REFLECT IT"S STATUS
  AFTER IT HAS BEEN WIDENED.

#
   ERROR := IF REFS1 OF MODE +
                REFS2 OF MODE +
                ROWS  OF MODE = 0
              THEN STRING S = SIMPLEMODE OF MODE;
                   SIMPLEMODE OF MODE :=
                IF S = "INT"
                  THEN "REAL"
                  ELIF S = "REAL"
                    THEN "COMPL"
                    ELIF S = "BITS"
                      THEN ROWS OF MODE := 1;
                           "BOOL"
                        ELIF S = "BYTES"
                          THEN ROWS OF MODE := 1;
                               "CHAR"
                  FI;
                  FALSE
              ELSE FALSE
            FI;
```

```
  PROC ROW = ( REF AMODE MODE, REF BOOL ERROR) VOID:

#

  THIS PROCEDURE UPDATES THE GIVEN MODE TO REFLECT IT"S STATUS
  AFTER THE MODE HAS BEEN ROWED.

#
   ERROR := IF ROWS OF MODE = 0
              THEN IF REFS1 OF MODE ¬= 0
                     THEN REFS2 OF MODE := REFS1 OF MODE;
                          REFS1 OF MODE := 0;
                          ROWS  OF MODE := 1
                     ELSE ROWS  OF MODE := 1
                   FI;
                   FALSE
              ELIF REFS1 OF MODE = 0
                THEN ROWS OF MODE +:= 1;
                      FALSE
                ELSE TRUE
            FI;
```

```
  PROC VOID =( REF AMODE MODE, REF BOOL ERROR) VOID:

#

  THIS PROCEDURE UPDATES THE GIVEN MODE TO REFLECT IT"S STATUS
  AFTER THE MODE HAS BEEN VOIDED.

#
    BEGIN
      IF SIMPLEMODE OF MODE = "PROC"
        THEN AMODE NEW := (Y_REFS1 OF MODE, Y_ROWS OF MODE,
                           Y_REFS2 OF MODE, SIMPLEMODE OF MODE,
                           0, 0, 0, 0, "");
             COERCE (MODE, NEW, "STRONG", LOC INT, LOC INT)
      FI;
      MODE := (0, 0, 0, "VOID", 0, 0, 0, 0, "")
    END;
  PROC BALANCE = ( (// ) AMODE UNITS,
                   AMODE MODE,
                   INT NBR_OF_MODES,
                   STRING SORT) VOID:

#

  THIS PROCEDURE WILL COMPUTE THE BALANCE MODE OF THE UNITS
  IN THE VECTOR CLAUSE.

#
    BEGIN
      BOOL MFLAG, ERROR,
      STRING SORT, REQUIRED_SORT,
      (// ) STRING SORT_REQUIRED = ("EMPTY", "SOFT", "WEAK",
                                   "STRONG", "STRONG", "STRONG") @ 0,
                  STRENGTH = ("SOFT", "WEAK", "MEEK", "FIRM", "STRONG"),
      (// ) INT STR_VAL = (1, 2, 2, 4, 5),
      INT LARGE, MINCONV;
      CALC_TARGET (UNITS, MODE, NBR_OF_MODES);
  RETRY:
      PRINT ((NEWLINE, "TARGET MODE"));
      PRINT_MODE (MODE);
      PRINT ((NEWLINE, NEWLINE, NEWLINE, "ATTEMPT BALANCE TO TARGET MODE"));
      MINCONV := 9;
      FOR I TO NBR_OF_MODES
        DO PRINT ((NEWLINE, NEWLINE, "UNIT-", I));
           COERCE (UNITS (/I/), MODE, "STRONG", LARGE, MFLAG);
           IF LARGE < 0 THEN FAILED FI;
           MINCONV := ( MINCONV < LARGE | MINCONV | LARGE)
        OD;
      PRINT ((NEWLINE, NEWLINE, NEWLINE));
      PRINT_MODE (MODE);
      PRINT ("  IS THE MODE OF THE BALANCE");
      REQUIRED_SORT := SORT_REQUIRED (/MINCONV/);
      IF MFLAG AND REQUIRED_SORT = "SOFT"
        THEN REQUIRED_SORT := "WEAK"
      FI;
      PRINT ((NEWLINE, SORT_REQUIRED, " WAS THE REQUIRED STRENGTH",
              NEWLINE, SORT, " WAS THE AVAILABLE STRENGTH"));
      FOR I TO UPB STRENGTH
        DO
```

```
        IF SORT = STRENGTH (/I/)
          THEN
            IF STR_VAL (/I/) < MINCONV
              THEN FAILED
              ELSE PRINT ((NEWLINE, "BALANCE VALID"));
                   RETURN
            FI
        FI
      OD;
    PRINT ((NEWLINE, "INVALID SORT"));
    RETURN;
FAILED:
    IF REFS1 OF MODE > 0
      THEN DEREFERENCE (MODE, ERROR)
      ELIF SIMPLEMODE OF MODE = "PROC"
        THEN DEPROCEDURE (MODE, ERROR)
        ELSE PRINT ((NEWLINE, "TARGET MODE CANNOT BE COERCED"));
             RETURN
    FI;
    IF ERROR
      THEN PRINT ((NEWLINE, "TARGET MODE CANNOT BE COERCED));
           RETURN
    FI;
    RETRY;
RETURN: SKIP
  END;
```

```
  PROC CALC_TARGET = ( (/ /) AMODE CLAUSE,
                       AMODE MODE,
                       INT NBR_OF_MODES) VOID:

#

  THIS PROCEDURE WILL CALCULATE A POSSIBLE BALANCE MODE GIVEN
  A VECTOR OF MODES TO BE BALANCED.

#
    BEGIN
      (/ /) SIRING SIMPLE_PRIOR = ("SKIP", "PROC", "FILE, "FORMAT",
                                   "BYTES", "CHAR", "STRING", "BITS",
                                   "BOOL", "INT", "REAL", "COMPL") @ 0,
      INT PRIOR_VAL, PRIOR_VAL2, I, J,
      BOOL PROC_SW := FALSE,
      PROC LOOK_UP = ( SIRING SIMPLEMODE) INT:
        BEGIN
          INT I := 0, J;
          WHILE SIMPLE_PRIOR (/ I /) ¬= SIMPLEMODE
            DO I +:= 1;
               IF I > UPB SIMPLE_PRIOR
                 THEN J := 0;
                      RETURN
               FI
            OD;
          J := I;
        RETURN:
          J
        END;
      PROC MAX = ( (/ /) INT SET) INT:

#

  THIS PROCEDURE COMPUTES THE LARGEST INTEGER IN A VECTOR.

#
        BEGIN
          INT MVAL := SET (/LWB SET/);
          FOR I FROM LWB SET + 1 TO UPB SET
            DO
              IF SET (/I/) > MVAL
                THEN MVAL := SET (/I/)
              FI
            OD;
          MVAL
        END;
      PROC MIN = ( (/ /) INT SET) INT:

#

  THIS PROCEDURE COMPUTES THE SMALLEST INTEGER IN A VECTOR.

#
        BEGIN
          INT MVAL := SET (/LWB SET/);
          FOR I FROM LWB SET + 1 TO UPB SET
            DO
              IF SET (/I/) < MVAL
```

```
            THEN MVAL := SET (/I/)
          FI
        OD;
        MVAL
    END;
  J := 1;
  WHILE SIMPLEMODE OF CLAUSE (/J/) = "SKIP"
    DO
     J +:= 1;
     IF J > NBR_OF_MODES
       THEN MODE := CLAUSE (/1/);
            RETURN
     FI;
  MODE := CLAUSE (/J/);
  IF SIMPLEMODE OF MODE = "PROC" THEN PROC_SW := TRUE FI;
  PRIOR_VAL := LOOK_UP (SIMPLEMODE OF MODE);
  PRIOR_VAL2 := LOOK_UP (Y_SIMPLEMODE OF MODE);
  FOR I FROM J + 1 TO NBR_OF_MODES
    DO
      IF SIMPLEMODE OF CLAUSE (/I/) ¬= "SKIP"
        THEN IF SIMPLEMODE OF MODE = "PROC" THEN PROC_SW := TRUE FI;
             REFS1 OF MODE := MIN ((REFS1 OF MODE,
                                    REFS1 OF CLAUSE (/I/)));
             REFS2 OF MODE := MIN ((REFS2 OF MODE,
                                    REFS2 OF CLAUSE (/I/)));
             ROWS OF MODE := MAX ((ROWS OF MODE,
                                   ROWS OF CLAUSE (II/)));
             Y_REFS1 OF MODE := MIN ((Y_REFS1 OF MODE,
                                      Y_REFS1 OF CLAUSE (/I/)));
             Y_REFS1 OF MODE := MIN ((Y_REFS2 OF MODE,
                                      Y_REFS2 OF CLAUSE (/I/)));
             Y_ROWS OF MODE := MAX ((Y_ROWS OF MODE,
                                     Y_ROWS OF CLAUSE (/I/)));
              NBR_OF_PROC_LEVELS OF MODE := MIN ((
                                 NBR_OF_PROC_LEVELS OF MODE,
                                 NBR_OF_PROC_LEVELS OF CLAUSE (/I/)));
             PRIOR_VAL := MAX ((PRIOR_VAL,
                                LOOK_UP (SIMPLEMODE OF CLAUSE (/I/))));
             PRIOR_VAL2 := MAX ((PRIOR_VAL2,
                                 LOOK_UP (Y_SIMPLEMODE OF CLO
                                 LOOK_UP (Y_SIMPLEMODE OF CLAUSE (/I/))))
      FI
    OD;
    SIMPLEMODE OF MODE := SIMPLE_PRIOR (/PRIOR_VAL/);
    IF PRIOR_VAL ¬= 0
      THEN Y_SIMPLEMODE OF MODE := SIMPLE_PRIOR (/PRIOR_VAL2/);
    FI;
    IF PROC_SW AND SIMPLEMODE OF PROC ¬= "PROC"
      THEN ROWS OF MODE := MAX ((ROWS OF MODE, Y_ROWS OF MODE));
           REFS1 OF MODE := MIN ((REFS1 OF MODE, Y_REFS1 OF MODE));
           REFS2 OF MODE := MIN ((REFS2 OF MODE, Y_REFS2 OF MODE));
           PRIOR_VAL := MAX ((PRIOR_VAL, PRIOR_VAL2));
           SIMPLEMODE OF MODE := SIMPLE_PRIOR (/PRIOR_VAL/);
           Y_SIMPLEMODE OF MODE := "";
           Y_ROWS OF MODE := 0
    FI;
RETURN:
    SKIP
END;
```

```
  PROC ASSIGN = ( REF AMODE M1, M2) VOID:

#

  THIS PROCEDURE WILL PERFORM THE COERCIONS NECESSARY TO PERFORM
  ASSIGNMENTS.

#
    BEGIN
      INT L;
      WHILE REFS1 OF M1 + REFS2 OF M1 + ROWS OF M1 = 0 AND
            SIMPLEMODE OF M1 = "PROC"
        DO DEPROCEDURE (M1, LOC BOOL);
           PRINT ((NEWLINE, "LHS DEPROCEDURED TO"));
           PRINT_MODE (M1)
        OD;
      REFS1 OF M2 +:= 1;
      COERCE (M1, M2, "STRONG", L, LOC BOOL);
      PRINT ((NEWLINE, (L = -1 | "ASSIGNMENT FAILED" | "ASSIGNMENT MADE")
    END;

#

  THIS IS THE MAIN PROGRAM WHICH READS ALL INPUT VALUES, INVOKES
  PARSE WHICH CONVERTS MODE REPRESENTATIONS TO INTERNAL FORM,
  AND INVOKES THE PROCEDURE INDICATED BY INPUT PARAMETER

#
AMODE MODE_1, MODE_2,
(/10/) AMODE CLAUSE,
STRING SYMBOL, STRING, SORT,
INT NBR_OF_MODES;
DO
  READ ((NEWLINE,STRING));
  IF STRING = "COERCE"
    THEN  READ ((NEWLINE,STRING));
          PRINT ((NEWLINE, NEWLINE, NEWLINE
                  "APRIORI MODE  ",STRING));
          PARSE (STRING,MODE_1);
          READ ((NEWLINE,STRING));
          PRINT ((NEWLINE, "APOSTERIORI MODE  ", STRING));
          PARSE (STRING, MODE_2);
          READ ((NEWLINE, SORT));
          PRINT ((NEWLINE, "SORT  ",SORT));
          COERCE (MODE_1, MODE_2, SORT, LOC INT, LOC BOOL)
    ELIF STRING = "BALANCE"
      THEN  READ ((NEWLINE, SORT));
            READ (NBR_OF_MODES);
            NEWPAGE (STANDOUT);
            FOR I TO NBR_OF_MODES
              DO
                READ ((NEWLINE, STRING));
                PRINT ((NEWLINE, "UNIT NUMBER-", I, STRING));
                PARSE (CLAUSE(/I/), STRING)
              OD;
              BALANCE (CLAUSE, LOC AMODE, NBR_OF_MODES, SORT)
      ELIF STRING = "ASSIGN"
        THEN  READ ((NEWLINE, STRING));
              PRINT ((NEWLINE, "ASSIGNMENT TO BE PERFORMED  ",
```

```
                    STRING));
            INT I := INDEX (STRING, ":=");
            PARSE (STRING(/1,I-1/) + " ",MODE_1);
            PARSE (STRING(/I+2, /), MODE_2);
            ASSIGN (MODE_1, MODE_2)
        ELSE PRINT ((NEWLINE, "INVALID COMMAND"));
            STOP
    FI
  OD
END
```

APPENDIX C

SAMPLE OUTPUT OF THE MODE PROCESSING ALGORITHMS

```
ASSIGNMENT TO BE PERFORMED--->REF REAL := REAL
MODES MATCH
ASSIGNMENT MADE


ASSIGNMENT TO BE PERFORMED--->REF REAL := REF REAL
DEREFERENCE
MODE AFTER COERCION REAL
MODES MATCH
ASSIGNMENT MADE


ASSIGNMENT TO BE PERFORMED--->REF REF REAL := REF REAL
MODES MATCH
ASSIGNMENT MADE


ASSIGNMENT TO BE PERFORMED--->REAL := REF REAL
DEREFERENCE
MODE AFTER COERCION REAL
NO POSSIBLE COERCION
ASSIGNMENT FAILED


ASSIGNMENT TO BE PERFORMED--->REAL := REAL
NO POSSIBLE COERCION
ASSIGNMENT FAILED


ASSIGNMENT TO BE PERFORMED--->REF REAL := PROC REF INT
DEPROCEDURE
MODE AFTER COERCION REF INT
DEREFERENCE
MODE AFTER COERCION INT
WIDEN
MODE AFTER COERCION REAL
MODES MATCH
ASSIGNMENT MADE


ASSIGNMENT TO BE PERFORMED--->PROC REF REF REAL := PROC REF REAL
LHS DEPROCEDURED TO REF REF REAL
DEPROCEDURE
MODE AFTER COERCION REF REAL
MODES MATCH
ASSIGNMENT MADE


ASSIGNMENT TO BE PERFORMED--->REF REF CHAR := CHAR
NO POSSIBLE COERCION
ASSIGNMENT FAILED
```

```
UNIT NUMBER- 1 REAL
UNIT NUMBER- 2 INT
UNIT NUMBER- 3 COMPL
TARGET MODE   COMPL


ATTEMPT BALANCE TO TARGET MODE

UNIT- 1
WIDEN
MODE AFTER COERCION COMPL
MODES MATCH

UNIT- 2
WIDEN
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
MODES MATCH

UNIT- 3
MODES MATCH


COMPL   IS THE MODE OF THE BALANCE
EMPTY  WAS THE REQUIRED STRENGTH
STRONG WAS THE AVAILABLE STRENGTH
BALANCE VALID
```

```
UNIT NUMBER- 1 REF (,) INT
UNIT NUMBER- 2 PROC () REF REAL
UNIT NUMBER- 3 REF REF () REF REF COMPL
TARGET MODE   (,) COMPL


ATTEMPT BALANCE TO TARGET MODE

UNIT- 1
DEREFERENCE
MODE AFTER COERCION (,) INT
NO POSSIBLE COERCION
TARGET MODE CANNOT BE COERCED
```

```
UNIT NUMBER- 1 REF REAL
UNIT NUMBER- 2 REF INT
UNIT NUMBER- 3 REF COMPL
TARGET MODE   REF COMPL


ATTEMPT BALANCE TO TARGET MODE

UNIT- 1
DEREFERENCE
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
NO POSSIBLE COERCION
TARGET MODE   COMPL


ATTEMPT BALANCE TO TARGET MODE

UNIT- 1
DEREFERENCE
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
MODES MATCH

UNIT- 2
DEREFERENCE
MODE AFTER COERCION INT
WIDEN
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
MODES MATCH

UNIT- 3
DEREFERENCE
MODE AFTER COERCION COMPL
MODES MATCH


COMPL   IS THE MODE OF THE BALANCE
MEEK    WAS THE REQUIRED STRENGTH
WEAK    WAS THE AVAILABLE STRENGTH
TARGET MODE CANNOT BE COERCED
```

```
UNIT NUMBER- 1 REF PROC REF REAL
UNIT NUMBER- 2 () COMPL
UNIT NUMBER- 3 PROC PROC INT
TARGET MODE   () COMPL

ATTEMPT BALANCE TO TARGET MODE

UNIT- 1
DEREFERENCE
MODE AFTER COERCION PROC  REF REAL
DEPROCEDURE
MODE AFTER COERCION REF REAL
DEREFERENCE
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
ROW
MODE AFTER COERCION () COMPL
MODES MATCH

UNIT- 2
MODES MATCH

UNIT- 3
DEPROCEDURE
MODE AFTER COERCION PROC  INT
DEPROCEDURE
MODE AFTER COERCION INT
WIDEN
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
ROW
MODE AFTER COERCION () COMPL
MODES MATCH

() COMPL   IS THE MODE OF THE BALANCE
EMPTY  WAS THE REQUIRED STRENGTH
STRONG WAS THE AVAILABLE STRENGTH
BALANCE VALID
```

```
UNIT NUMBER- 1 REF INT
UNIT NUMBER- 2 REF REAL
UNIT NUMBER- 3 PROC () REAL
UNIT NUMBER- 4 PROC () INT
TARGET MODE   REAL

ATTEMPT BALANCE TO TARGET MODE

UNIT- 1
DEREFERENCE
MODE AFTER COERCION INT
WIDEN
MODE AFTER COERCION REAL
MODES MATCH

UNIT- 2
DEREFERENCE
MODE AFTER COERCION REAL
MODES MATCH

UNIT- 3
DEPROCEDURE
MODE AFTER COERCION () REAL
NO POSSIBLE COERCION
TARGET MODE CANNOT BE COERCED
```

```
UNIT NUMBER- 1 REF BOOL
UNIT NUMBER- 2 SKIP
TARGET MODE   REF BOOL


ATTEMPT BALANCE TC TARGET MODE

UNIT- 1
MODES MATCH

UNIT- 2
MODE SKIP COERCED TO REF BOOL


REF BOOL    IS THE MODE OF THE BALANCE
EMPTY  WAS THE REQUIRED STRENGTH
FIRM   WAS THE AVAILABLE STRENGTH
BALANCE VALID


A PRIORI MODE  PROC PROC REF REAL
APOSTERIORI MODE  VOID
SORT  STRONG
DEPROCEDURE
DEPROCEDURE
VOID
MODE AFTER COERCION VOID
MODES MATCH


A PRIORI MODE  PROC REAL
APOSTERIORI MODE  REAL
SORT  WEAK
DEPROCEDURE
MODE AFTER COERCION REAL
MODES MATCH


A PRIORI MODE  PROC PROC REAL
APOSTERIORI MODE  PROC REAL
SORT  WEAK
DEPROCEDURE
MODE AFTER COERCION PROC  REAL
MODES MATCH


A PRIORI MODE  PROC PROC REAL
APOSTERIORI MODE  REAL
SORT  WEAK
DEPROCEDURE
MODE AFTER COERCION PROC  REAL
DEPROCEDURE
MODE AFTER COERCION REAL
MODES MATCH
```

```
A PRIORI MODE  REF PROC REAL
APOSTERIORI MODE  PROC REAL
SORT  WEAK
NO POSSIBLE COERCION


A PRIORI MODE  REF REF REAL
APOSTERIORI MODE  REF REAL
SORT  SOFT
DEREFERENCE
MODE AFTER COERCION REF REAL
MODES MATCH


A PRIORI MODE  REF REAL
APOSTERIORI MODE  REAL
SORT  SOFT
NO POSSIBLE COERCION


A PRIORI MODE  PROC REF REF REAL
APOSTERIORI MODE  REF REAL
SORT  SOFT
DEPROCEDURE
MODE AFTER COERCION REF REF REAL
DEREFERENCE
MODE AFTER COERCION REF REAL
MODES MATCH


A PRIORI MODE  REF REF REF REAL
APOSTERIORI MODE  REF REAL
SORT  SOFT
DEREFERENCE
MODE AFTER COERCION REF REF REAL
DEREFERENCE
MODE AFTER COERCION REF REAL
MODES MATCH


A PRIORI MODE  PROC PROC REAL
APOSTERIORI MODE  REAL
SORT  SOFT
DEPROCEDURE
MODE AFTER COERCION PROC  REAL
DEPROCEDURE
MODE AFTER COERCION REAL
MODES MATCH


A PRIORI MODE  REF REF REAL
APOSTERIORI MODE  REAL
SORT  SOFT
```

```
DEREFERENCE
MODE AFTER COERCION REF REAL
NO POSSIBLE COERCION


A PRIORI MODE  REF REAL
APOSTERIORI MODE  REAL
SORT  SOFT
NO POSSIBLE COERCION


A PRIORI MODE  REF REF REAL
APOSTERIORI MODE  REF REAL
SORT  MEEK
DEREFERENCE
MODE AFTER COERCION REF REAL
MODES MATCH


A PRIORI MODE  REF REAL
APOSTERIORI MODE  REAL
SORT  MEEK
DEREFERENCE
MODE AFTER COERCION REAL
MODES MATCH


A PRIORI MODE  PROC REF REF REAL
APOSTERIORI MODE  REF REAL
SORT  MEEK
DEPROCEDURE
MODE AFTER COERCION REF REF REAL
DEREFERENCE
MODE AFTER COERCION REF REAL
MODES MATCH


A PRIORI MODE  REF REF REF REAL
APOSTERIORI MODE  REF REAL
SORT  MEEK
DEREFERENCE
MODE AFTER COERCION REF REF REAL
DEREFERENCE
MODE AFTER COERCION REF REAL
MODES MATCH


A PRIORI MODE  PROC PROC REAL
APOSTERIORI MODE  REAL
SORT  MEEK
DEPROCEDURE
MODE AFTER COERCION PROC  REAL
DEPROCEDURE
MODE AFTER COERCION REAL
```

```
MODES MATCH


A PRIORI MODE  REF REF REAL
APOSTERIORI MODE  REAL
SORT  MEEK
DEREFERENCE
MODE AFTER COERCION REF REAL
DEREFERENCE
MODE AFTER COERCION REAL
MODES MATCH


A PRIORI MODE  REF REAL
APOSTERIORI MODE  REAL
SORT  MEEK
DEREFERENCE
MODE AFTER COERCION REAL
MODES MATCH


A PRIORI MODE  REF PROC REF INT
APOSTERIORI MODE  (,) COMPL
SORT  STRONG
DEREFERENCE
MODE AFTER COERCION PROC  REF INT
DEPROCEDURE
MODE AFTER COERCION REF INT
DEREFERENCE
MODE AFTER COERCION INT
WIDEN
MODE AFTER COERCION REAL
WIDEN
MODE AFTER COERCION COMPL
ROW
MODE AFTER COERCION () COMPL
ROW
MODE AFTER COERCION (,) COMPL
MODES MATCH


PROGRAM IS STOPPED.
```

APPENDIX D

A GRAMMAR FOR THE LANGUAGE ACCEPTED BY THE OKLAHOMA STATE UNIVERSITY ALGOL 68 COMPILER

The following is a modified Backus-Naur Form of a grammar which generates the language which is accepted by the Oklahoma State University ALGOL 68 Compiler after the features described in this thesis have been implemented. The grammar is expressed by rules of the form:

<META SYMBOL> ::= def

this rule is read "<META SYMBOL> is defined to be (:::=) def." If there are several definitions of the same meta symbol they may be combined into one rule by the "|" symbol read "or is a" such as:

<META SYMBOL> :::= def 1

<META SYMBOL> :::= def 2

becomes

<META SYMBOL> :::= def 1 | def 2.

The definition of a meta symbol may contain any sequence of meta symbols or terminal symbols (symbols without brackets). The special symbol ".EMPTY" means that the meta symbol may be replaced by the empty string. The goal rule for the grammar is <PROGRAM>. If the symbols "|", "<" or ">" are required as terminal symbols they will be enclosed within quotation marks.

```
<PROGRAM>::=                <ENCLOSED CLAUSE>

<ENCLOSED CLAUSE>::=        BEGIN <SERIAL CLAUSE> END|
                            BEGIN <COLLATERAL CLAUSE> END|
                            <CONDITIONAL CLAUSE>|
                            <CASE CLAUSE>|
                            <LOOP CLAUSE>

<SERIAL CLAUSE>::=          <UNIT DEC LIST1><L UNIT LIST1>

<UNIT DEC LIST1>::=         <UNIT>;<UNIT DEC LIST2>|
                            <DECLARATIONS>;<UNIT DEC LIST2>|
                            .EMPTY

<UNIT DEC LIST2>::=         <UNIT>;<UNIT DEC LIST2>|
                            <DECLARATIONS>;<UNIT DEC LIST2>|
                            <UNIT>|
                            <DECLARATIONS>

<L UNIT LIST1>::=           <L UNIT><L UNIT LIST2>|
                            <L UNIT>

<L UNIT LIST2>::=           ;<L UNIT><L UNIT LIST2>|
                            EXIT<ID>:<UNIT><L UNIT LIST2>|
                            ;<L UNIT>|
                            EXIT<ID>:<UNIT>

<L UNIT>::=                 <ID>:<L UNIT>|
                            <UNIT>

<COLLATERAL CLAUSE>::=      <UNIT>,<UNIT COMMA LIST>

<UNIT COMMA LIST>::=        <UNIT>,<UNIT COMMA LIST>|
                            <UNIT>

<CONDITIONAL CLAUSE>::=     IF <SERIAL CLAUSE> THEN <SERIAL CLAUSE><COND END>

<COND END>::=               <ELIF PART LIST><ELSE PART> FI

<ELIF PART LIST>::=         ELIF <SERIAL CLAUSE> THEN <SERIAL CLAUSE>
                            <ELIF PART LIST>|
                            .EMPTY

<ELSE PART>::=              ELSE <SERIAL CLAUSE>|
                            .EMPTY

<CASE CLAUSE>::=            CASE <SERIAL CLAUSE> IN <COLLATERAL CLAUSE><CASE END>

<CASE END>::=               <OUSE PART LIST><OUT PART> ESAC

<OUSE PART LIST>::=         OUSE <SERIAL CLAUSE> IN <COLLATERAL CLAUSE>
                            <OUSE PART LIST>|
                            .EMPTY

<OUT PART>::=               OUT <SERIAL CLAUSE>|
                            .EMPTY

<LOOP CLAUSE>::=            <FOR PART><FROM PART><BY PART><TO PART><WHILE PART>
                            DO <SERIAL CLAUSE> OD
```

```
<FOR PART>::=              FOR <ID>|
                           .EMPTY

<FROM PART>::=             FROM <UNIT>|
                           .EMPTY

<BY PART>::=               BY <UNIT>|
                           .EMPTY

<TO PART>::=               TO <UNIT>|
                           .EMPTY

<WHILE PART>::=            WHILE <SERIAL CLAUSE>|
                           .EMPTY

<DECLARATIONS>::=          <DECLARATION>,<DECLARATIONS>|
                           <DECLARATION>

<DECLARATION>::=           <MODE DECLARATION>|
                           <IDENTITY DECLARATION>|
                           <VARIABLE DECLARATION>|
                           <PROC DECLARATION>

<MODE DECLARATION>::=      MODE <MODE INDICANT>=<MODE>,<MODE DECLARATION>|
                           MODE <MODE INDICANT>=<MODE>

<IDENTITY DECLARATION>::=<MODE><IDENT INIT LIST>

<IDENT INIT LIST>::=       <ID>=<UNIT>,<IDENT INIT LIST>|
                           <ID>=<UNIT>

<VARIABLE DECLARATION>::=<MODE><VAR INIT LIST>

<VAR INIT LIST>::=         <ID>:=<UNIT>,<VAR INIT LIST>|
                           <ID>:=<UNIT>

<PROC DECLARATION>::=      PROC <ID><TYPE INIT><ROUTINE TEXT>

<TYPE INIT>::=             =|
                           :=

<MODE>::=                  <REF LIST><ROW LIST><REF LIST><BASIC MODE>

<REF LIST>::=              REF <REF LIST>1
                           .EMPTY

<ROW LIST>* ::=            <ROW><ROW LIST>|
                           .EMPTY

<ROW>::=                   (/ <UNIT> /)|
                           (/ <UNIT>:<UNIT> /)|
                           (/ /)

<MODE INDICANT>::=         <MODE INDICATION>

<BASIC MODE>::=            INT|REAL|COMPL|CHAR|BOOL|
                           <MODE INDICATION>|
                           <P DEC>
```

```
<P DEC>::=                  PROC <VIRTUAL PARAMETERS><MODE>

<VIRTUAL PARAMETERS>::=     ( <MODE LIST> ) |
                            .EMPTY

<MODE LIST>::=              <MODE>,<MODE LIST>|
                            <MODE>

<UNIT>::=                   <ASSIGNATION>|
                            <ROUTINE TEXT>|
                            <IDENTITY RELATION>|
                            <JUMP>|
                            SKIP|
                            <TERTIARY>

<ASSIGNATION>::=            <TERTIARY>  :=  <UNIT>

<ROUTINE TEXT>::=           <FORMAL PARAMETERS><MOID>:<UNIT>

<FORMAL PARAMETERS>::=      ( <FORM PARM> )|
                            .EMPTY

<FORM PARM>::=              <MODE SET>,<FORM PARM>|
                            <MODE SET>

<MODE SET>::=               <MODE><ID LIST>

<ID LIST>::=                <ID>,<ID LIST>|
                            <ID>

<MOID>::=                   <MODE>|
                            VOID

<IDENTITY RELATION>::=      <TERTIARY>:=:<TERTIARY>|
                            <TERTIARY>:¬=:<TERTIARY>

<JUMP>::=                   <GO TO OPTION><ID>

<GO TO OPTION>::=           GO TO

<TERTIARY>::=               <FORMULA>|
                            NIL

<FORMULA>::=                <FORMULA><DYADIC OPERATOR><FORMULA>|
                            <FORMULA>|
                            <MONADIC OPERATOR><FORMULA>

<SECONDARY>::=              RE <PRIMARY>|
                            IN <PRIMARY>|
                            <PRIMARY>

<PRIMARY>::=                ID | DENOTATION | FORMAT_TEXT |
                            <CALL>|<CAST>|<SLICE>|<ENCLOSED CLAUSE>

<CALL>::=                   <PRIMARY><ACTUAL PARAMETERS>

<ACTUAL PARAMETERS>::=      ( <UNIT COMMA LIST> )|
                            .EMPTY

<CAST>::=                   <MOID> ( <UNIT> )
```

```
<SLICE>::=              <PRIMARY>  <SLICER LIST>

<SLICER LIST>::=        <SLICER>,<SLICER LIST>|
                        <SLICER>

<SLICER>::=             <FROM UNIT><UP TO UNIT><AT UNIT>

<FROM UNIT>::=          <UNIT>|
                        .EMPTY

<UP TO UNIT>::=         : <UNIT>|
                        :|
                        .EMPTY

<AT UNIT>::=            @ <UNIT>|
                        .EMPTY
```

*SUCCESSIVE ROW ENTITIES MAY BE COMBINED INTO ONE SET OF ⟨/ AND /⟩ SYMBOLS BY SEPARATING THE INNER PARTS BY COMMAS.
EXAMPLES: (/X/) (/0:2X/) MAY BE WRITTEN AS (/X,0:2X/) OR (/ /) (/ /) MAY BE WRITTEN AS (/,/).

APPENDIX E

USER'S GUIDE

Control Cards

A description of the ;JOB CARD options and format can be found on page 37 of the thesis by Jensen (1). At this time the revised version of the compiler is operational only on the IBM 360/65 running under OS/MVT. The job control language required to execute the revised version is shown in Figure 44.

Restrictions

Beginning on page 38 Jensen (1) lists ten restrictions upon the ALGOL 68 subset which he implemented. The following set of restrictions includes those restrictions of Mr. Jensen which are still applicable and also all restrictions upon the newly implemented facilities.

1) All ALGOL 68 keywords are reserved;
2) Keywords must be separated from identifiers, denotations and other keywords by at least one blank;
3) Keywords, multiple symbol operators and denotations may not contain embedded blanks (except of course ⌜_⌟ CHAR denotations);
4) All identifiers of non-procedured modes must be declared before they are referenced in order to produce predictable results;
5) Identifiers may not contain embedded blanks, but the break character (_) may be used to improve readability.

```
//JOBNAME   JOB (XXXXX,XXX-XX-XXXX,X),CLASS=B
/*SETUP   DISK=1
/*PASSWORD XXXX
//STEPNAME  EXEC  PGM=ALGOL,REGION=140K
//STEPLIB  DD  DSN=COMSC.PART.SEAY.LOAD,UNIT=2314,
//              VOL=SER=DISK28,DISP=SHR
//FT03F001  DD  DSN=COMSC.SEAY.ERROR.A68,
//               VOL=SER=DISK28,UNIT=2314,DISP=SHR
//FT06F001  DD  SYSOUT=A
//FT05F001  DD  *
     :JOB
          .
          .
          .
         ALGOL 68 SOURCE PROGRAM
          .
          .
          .
     :ENTRY
          .
          .
          .
         STANDIN DATA (IF ANY)
          .
          .
          .
     :IBSYS
//
```

Figure 44. Job Control Language for IBM 360/65 Execution

6) Row displays are limited to one row only, such as, (1,2,3);

7) Balancing is not performed for row displays; therefore, the mode of the balance is assumed

to be of the same mode as its first unit (the mode of the first unit may of course always be modified by way of a cast);

8) Objects of mode ROW^{i} REF^{j} amode may not be used in transput operations;

9) Any mode indications used in the program must be defined before use;

10) Rowing may only be used to create REF [ ] amode values from amode values or REF amode values;

11) A program may not contain more than 25 different ROW displays.

New Features

Procedure Variables

There are two methods of declaring procedure variables and constants. They may be declared as outlined by Eyler (3) in his thesis on procedure implementation, or as an alternative the programmer may write the following type declaration:

PROC VIRTUAL_PARAMETERS_PACK YIELDING_MOID ID_INIT_LIST.

This declares each of the identifiers contained in the identifier list to be a procedure variable of the mode indicated by the virtual parameters pack and yielding mode field. These variables may be assigned routines (of a suitable mode) dynamically during the elaboration of the program. The ID_INIT_LIST also allows procedure variables to be initialized by units yielding the proper mode within the

declaration. Figure 45 shows some examples of the types of procedure declarations currently allowed.

```
     Procedure constant:
PROC a = (REAL x,y) REAL: (x - y)/(x + y) * 100.0;
     Procedure variable initialized by a routine:
PROC b:= (REAL x,y) REAL: (x - y)/(x + y) * 100.0;
     Procedure variable assigned a routine at a
     later time:
PROC (REAL,REAL) REAL c;
  .
  .
  .
c:= (REAL x,y) REAL: (x - y)/(x + y) * 100.0;
     Procedure variable initialized by a unit:
PROC (REAL,REAL) REAL d:=
     IF p THEN (REAL x,y) REAL: (x - y)/(x + y) * 100.0
          ELSE (REAL x,y) REAL: (y - x)/(x + y) * 100.0
     FI;
```

Figure 45. Sample Procedure Declarations

REF Amode Variables (pointers)

The compiler does not currently support structures or list processing. Pointer variables have only a limited usefullness under these restrictions. Two valid uses of pointer variables are:

1) to achieve the effect of CALL_BY_REFERENCE

parameters (all ALGOL 68 parameters are CALL_BY_VALUE) and

2) if it is known that a particular element of an array is to be referenced more than once, pointer variables may save much processing time.

Figure 46 provides an example of the definition of pointer variables to decrease execution time. It also provides an example of mode declarations.

```
BEGIN
     MODE .TREE_NODE = [3] INT;
     REF .TREE_NODE CUR_NODE;
     [100] .TREE_NODE TREE;
     INT ROOT:= 0;
     PROC SEARCH = (INT ARGUMENT) REF .TREE_NODE:
     BEGIN
          REF .TREE_NODE X:= NIL;
          INT SRCH:= ROOT;
          WHILE SRCH:  =: NIL
          DO
               CUR_NODE:= TREE [SRCH,];
               REF INT LLINK:= CUR_NODE [1,];
                       KEY:= CUR_NODE [2,];
                       RLINK:= CUR_NODE [3];
               IF ARGUMENT  KEY THEN SRCH:= LLINK
                    ELIF ARGUMENT = KEY
                    THEN X:= CUR_NODE;
                         OUT_OF_PROC
                    ELSE SRCH:= RLINK
               FI
          OD;
          OUT_OF_PROC:  SKIP
     END
END
```

Figure 46. Example Program Illustrating Possible Uses of REF Amode Variables

APPENDIX F

SYSTEM PROGRAMMER'S GUIDE

Symbol Table Modification

Moving the Symbol Table From Memory to Disk

The current version of the compiler does not actually perform any disk input/output for intermediate files. A new version of subprogram ALGIO which saves each 80 word record of output in an array was written. When an input request is made, tables built during output operations are searched to locate the desired record. After the record has been located it is moved into the output parameter area and the subroutine is exited.

In order to allow the input/output files to actually be written onto disk it is necessary to provide the necessary JOB CONTROL LANGUAGE for each file. It is also necessary to replace the current version of ALGIO by the original version.

Modifying the Symbol Table Size

If it is found desirable to modify the symbol table size; it is necessary to change the data statement found in subprogram ALGZA to

```
DATA TBSZ/N/
```

where the N is replaced by the size (in words) desired for the symbol table (N should be an integral multiple of 80).

If the symbol table is on disk the only other changes required are the DEFINE FILE statement for file number 11

and of course changing the JOB CONTROL LANGUAGE specification for the file. If the symbol table file is located in memory, then it will be necessary to make the following changes to ALGIO.

1) The dimension statement for the variable DISK must be increased to reflect the total number of words expected for all intermediate files (DISK should be an integral multiple of 80);
2) The dimension statement for LOG must be changed to

   DIMENSION LOC (i,3)

   where i = DISK/80;
3) The data statement for NRPGS must be changed to the value DISK/80.

Operator Declarations

The subprogram ALGZO has been included with the modifications made to the compiler. ALGZO is not currently called by any existing routine. It has the function of inserting operator declarations into the symbol table. Figure 47 is a diagram which shows how the operator declarations would be placed in the symbol table by ALGZO. Figure 48 is a list of the formal parameters of subprogram ALGZO along with the meaning of each parameter. Operator declarations could be processed by treating it as a procedure declaration. The declaration OP + = (INT A,B) INT:~ could conceptually be parsed as

PROC TEMP_ID = (INT A,B) INT: ~ .

A symbol table entry would be made in the normal manner for the procedured mode identified by TEMP_ID, followed by a call to ALGZO to enter the routine definition into the operator routine list.

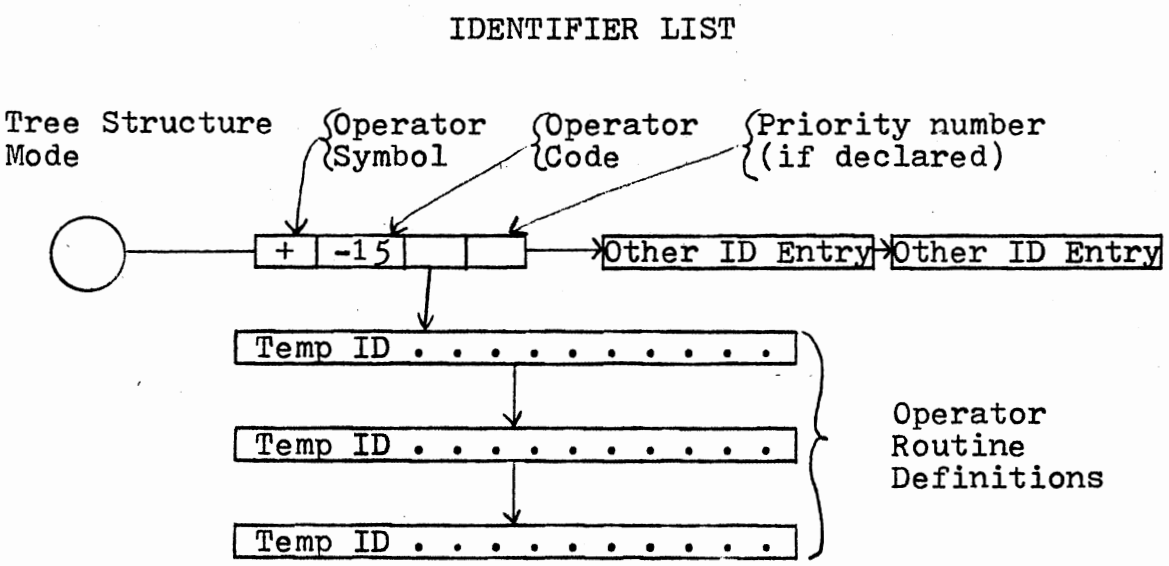


Figure 47. Operator Declaration Structure

Table I gives the functions of the subprograms that have been added to the compiler.

Formal Parameter	Use/Meaning
OPSYM	The internal code for the operator symbol being defined
OPND1	The mode number of the left operand of this operator
OPND2	The mode number of the right operand of this operator (use zero if operator is monodic)
YMOID	The mode number of the yielding value of the operator
ROUT	The temporary identifier assigned to this particular operator routine
PRIO	The priority to be assigned to all dyadic operators using this symbol (when PRIO is not zero the only input parameter values used are OPSYM and PRIO)
PTR	Return address of the symbol table node created as a result of this call
IER	Error code IER = 0 - No error IER = 1 - Attempt to insert two routines with same temporary identifier IER = 2 - Attempt to define a routine after a label in the current block IER = 3 - Duplicate priority number definition

Figure 48. Formal Parameters for Subprogram ALGZO

TABLE I

FUNCTIONS OF ADDED SUBPROGRAMS

Subprogram	Functions
Phase 3	
ALGF5	Performs DISK I/O for the Phase 3 pass through the source code (has the ability to access each word of the file directly by position).
Phase 3.5	
ALGF2	Fetches next input symbol for Phase 3.5. Input may come from the symbol table area (mode declarations on the input source file).
ALGIO	The incore storage version of the I/O routine.
ALGYA	Saves space in the symbol table area by packing up to seven one word entries into the eight word symbol table node.
ALGYB	Establishes the standard environment by loading the mode table. (Can also be modified to load standard operator definitions.)
ALGYC	Unpacks the data packed by ALGYA. Repeated calls to ALGYC will make all values in the list available.
ALGZA	Performs the paging necessary for access to the symbol table and through ALGIO performs any necessary input/output operations (see discussion on symbol table location).
ALGZB	Provides access to the symbol table area. Data may not be fetched or stored in such a way as to span two symbol table nodes.
ALGZC	Causes the tree structure pointer of the symbol table to be decreased by one nesting level.

TABLE I (Continued)

Subprogram	Functions
ALGZD	Increase the nesting level during symbol table construction.
ALGZF	Inserts an identifier into the identifier list associated with the current block.
ALGZG	Search the symbol table for an occurrence of an identifier.
ALGZH	Prints the attribute and cross reference listing (if requested), it also causes the mode table print routine to be called.
ALGZI	This subroutine blocks the symbols output from Phase 3.5. Source symbols are output as is; however, object code symbols are prefixed by a value equal to (1000 + number of object text words). Note: It is possible to have the value 1000 if it is necessary to complete a record and no object text would fit in the remaining space after the code symbol.
ALGZJ	Pushes values onto the compile time stack.
ALGZK	Pops values from the compile time stack.
ALGZL	Parses loop clauses.
ALGZM	Parses declarations.
ALGZN	Parses the mode indications.
ALGZS	Generates allocate storage instructions for declared variables.
ALGZU	Equivalences user defined modes.
ALGZV	Computes the nesting level for each symbol table node.
ALG3B	Main line Phase 3.5

TABLE I (Continued)

Subprogram	Functions
GETMD	Fetches the mode table entry for the indicated mode number.
INSMD	Inserts modes into the mode table (also computes related modes).
JTST	Tests switches to determine if debugging information is to be output.
MDTST	Allows tests of various fields of a mode table entry.
MODET	Performs the insertion of a single mode into the mode table.
PRTMD	Converts the coded values of a mode table entry into A1 characters suitable for printing.
PRTMT	Prints the entire mode table using subprogram PRTMD.
Phase 4	
ALGZE	Increase the symbol table level (assuming tree structure has been built).
BAL	Computes the balance mode for multiple completion clauses. (ALGBL generates the balancing code.)
COERC	Calculates the coercion path from the A PRIORI mode to the A POSTERIORI mode.
DREF4	Computes the mode number of the mode which has one less REF than the input mode (generates code if necessary).
POSS	Determines the set of all possible coercions for a given syntactic position.
ROW4	Completes the mode number of the mode which has one more row than the input mode (generates code if necessary).

TABLE I (Continued)

Subprogram	Functions
SIFT	Reduces the set determined in POSS to the unique coercion to be performed.
WIDE4	Computes the mode number for the mode which is one level wider than the input mode (generates code if necessary).
VOID	Computes the actions necessary to void the current mode (generates code if necessary).
Phase 5	
ALGYF	Implements new pseudo operation codes 901-907.
ALGYG	Implements and becomes operators (leaving a REF amode termporary on the stack top.

VITA

Walter Michael Seay

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF A SUBSET OF MODES IN AN ALGOL 68 COMPILER

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in San Diego, California, June 24, 1946, the son of Mr. and Mrs. Sidney E. Seay.

Education: Graduated from James Madison High School, San Diego, California, in June, 1964; attended San Diego Junior College, 1963-64; attended San Antonio College, 1965-66; received Bachelor of Science degree in Mathematics from Troy State University in 1974; completed requirements for the Master of Science at Oklahoma State University in July, 1976.

Professional Experience: Programming Supervisor and computer programmer, United States Air Force, 1964-73; part-time computer operator, Troy State University, 1973-74; graduate assistant, Computing and Information Sciences Department, Oklahoma State University, 1974-76; member of the Association for Computing Machinery.