IMPLEMENTATION OF A SCIENTIFIC SUBSET OF ALGOL 68

By

JOHN CLUTE JENSEN

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1971

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, 1973

IMPLEMENTATION OF A SCIENTIFIC SUBSET OF ALGOL 68

Thesis Approved:

H. E. Hedrick
Thesis Adviser

James R. Vondoran

D D Fisher

N N Durham
Dean of the Graduate College

PREFACE

This paper describes the implementation of a scientific subset of the ALGOL 68 programming language. It is concerned with the methods used to implement a translator system which generates code which is interpretively executed. The system is written in the basic FORTRAN language to allow machine independence.

I would like to express my gratitude to my major advisor, Dr. G. E. Hedrick, for his advice and guidance during this project. Also, appreciation is expressed to my other committee members, Dr. Donald D. Fisher and Dr. James R. Van Doren, for their suggestions and assistance in the preparation of this paper.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Objectives

The ALGOL 68 language is perhaps the most sophisticated programming language to be developed. Its designers have presented ALGOL 68 with a precise, although somewhat complicated, language definition in the "Report on the Algorithmic Language ALGOL 68." (18) Features in the language allow virtually limitless expansion of the language to include whatever facilities an installation might require.

Incorporated within the ALGOL 68 language are many of the desirable features of ALGOL 60, COBOL, FORTRAN and PL/I, along with some features which are unique to ALGOL 68. Since it is a relatively young language, ALGOL 68 translators are scarce. A need was seen for a translator which could be used as an instructional tool for teaching ALGOL 68, either by itself or as a transition from other programming languages. The translator would not need to be an implementation of the entire ALGOL 68 languange, but it should contain the basic constructs of the language.

To meet this purpose, a scientific subset of the ALGOL 68 language was selected for implementation. Contained within the subset are capabilities for performing operations on simple numerical and logical values, along with some limited character and string manipulations. This is sufficient to illustrate many of the basic properties of ALGOL 68 and leads to an understanding of the more advanced features of the language.

Also, the subset is complete enough to be used for many scientific applications. Detailed error messages are included to allow easier understanding of specific programming examples.

History of ALGOL 68

Background

The formal definition of ALGOL 68 was presented in a report to the General Assembly of the International Federation for Information Processing (IFIP) in 1968. The report was prepared by Working Group 2.1 on ALGOL and was reviewed by Technical Committee 2 on Programming Languages for publication. The Report was subsequently published in Numerische Mathematik in 1969.

Since the publication of the Report, (18) conferences have been held on the ALGOL 68 language and its implementation. Proposals have arisen from these conferences concerning changes of the language. Some modifications have been made but the language remains essentially unchanged from its original definition.

ALGOL 68 Definitions

The definitions of some essential ALGOL 68 terms are given below.

Coercion. An implicit change of the mode of an operand dependent upon syntactic position.

Denotation. An ALGOL 68 constant or literal value.

Descriptor. A data structure consisting of an offset and a set of quintuples which describe a row of an array.

Elaboration. Execution.

Environmental Enquiry. A pre-defined constant supplying information concerning a specific property of the implementation.

Multiple Value. A multiple value is composed of a descriptor and any number of rows.

Pragmat. An ALGOL 68 comment directing a particular implementation to take a specified action which lies outside the definition of the language.

Row. The elements of one dimension of an array.

Standard Prelude. A set of standard declarations which specify environmental enquiries, standard priorities (of operators), standard operations, standard mathematical constants or functions, or transput declarations.

Transput. Input and output.

Literature Review

Much of the information related to the implementation of ALGOL 68 comes out of the formal and informal conferences on the language. Many of the papers presented at these conferences were concerned with the theoretical aspects of the two level ALGOL 68 grammar and constructs dealing with the structure of ALGOL 68 programs. Others proposed solutions to applications programming problems by the use of ALGOL 68. Of the papers which dealt with practical implementation, a great many were concerned with topics which were beyond the scope of this project.

Of the existing ALGOL 68 implementations, the earliest and perhaps most noteworthy is the ALGOL 68-R implementation at the Royal Radar Establishment in Malvern, England. (4,5,19,20) The implementation is designed for the ICL 1907F computer. This one pass translator accepts a

large subset of the ALGOL 68 language and has extensive program library capabilities.

At the Technical University of Munich, another ALGOL 68 subset is being implemented. Hill, et al. (9) present a detailed description of the implementation techniques being used in this project. Use of this particular implementation will be limited since it is being written in machine language for the Telefunken TR 4 computer. This machine is not in common use.

Oklahoma State University has implemented a system which accepts an ALGOL 68 program and outputs PL/I source code which is equivalent to the ALGOL 68 program. (7) This system functions correctly for a scientific subset of ALGOL 68, but has the obvious disadvantage of having to compile the program twice.

In general, the implementations mentioned above have restricted ALGOL 68 transput to include only unformatted transmission of data. Berry (1,2,3) has implemented a system for formatted transput. This system parses formats at run time and uses interpretive execution to effect their elaboration.

Smith, et al. (15) have developed an algorithm for the recognition of ALGOL 68 denotations using a finite state automaton. Hedrick and Smith have published a comprehensive study of ALGOL 68 context conditions. (8)

Problem Statement

The purpose of this project is to implement a scientific subset of the computer programming language ALGOL 68. The language translator is written in IBM 1130 basic FORTRAN, (20) and it generates 4-tuples (quad-

ruples) of code which are interpretively executed, thus making the system machine independent. The implementation is intended for use as an introduction to the ALGOL 68 language.

CHPATER II

IMPLEMENTATION CONSIDERATIONS

IBM 1130 Implementation

Selecting a Small Computer

In order to offer a measure of portability, it was decided to implement the ALGOL 68 translator on a small computer. This insures that the translator is kept small due to the storage limitations of machine. Thus, it is possible to execute the translator on different systems which are the same size or larger. An IBM 1130 was available at Oklahoma State University, and it was suitable for this purpose.

System Configuration

The ALGOL 68 subset was implemented on an IBM 1130 computer with 8K words of core storage. The computer operates under Disk Moniter System Version 2, Modification 8, using an 1131 Single Disk Storage unit. (12) The principal input device is a 1442 card read/punch, while the principal output device is the console typewriter.

Storage Considerations

Limited core storage presented the most serious problem in implementing the subset. The compiler had to be written in four distinct phases with the interpretive executor as a fifth phase. As the execution

of each phase is completed, a LINK instruction is issued to signal the core load builder to load and begin execution of the next phase.

Efforts to conserve storage resulted in the use of data handling techniques which require a small amount of core storage. This does, however, result in increased execution time. This is particularly evident in table lookups, where linear searches are used.

Data areas were frequently redefined with EQUIVALENCE statements to permit reuse of storage and more efficient access. The FORTRAN compiler on the 1130 does no subscript optimization for array references with constant subscripts. Therefore, equivalencing a specific array element to a simple variable name causes a direct, rather than indexed, reference in the FORTRAN object code.

Although the 1130 core load builder is capable of creating core image programs with dynamic overlays, this feature was avoided whenever possible. The load on call, or LOCAL, facility allows shared use of core storage by infrequently called subroutines, but significantly increases execution time due to disk accesses. (12) Also, the system routine which loads LOCALed subroutines is quite large and in itself causes storage problems. Only in the parsing and code generation phase of the compiler and in the interpretive executor are LOCAL subroutines used. Storage requirements for these routines could not be met using other methods.

1130 Problems

Of the problems particular to the 1130, the first to occur during this implementation was that of representing special characters. The 1130 FORTRAN compiler recognizes only a subset of the EBCDIC character

set. Therefore, some symbols which are an integral part of ALGOL 68 (e.g., :, ;, #, ¢) must be defined with hexidecimal or integer equivalents of the FORTRAN A1 EBCDIC values. Even when special characters are correctly defined within the scanner, the standard I/O routines cannot print them. To overcome this problem, the compiler formats its own print lines and calls an assembler language program to perform the output. Besides supporting the full EBCDIC character set, this routine is considerably smaller than the FORTRAN formatting routines and permits overlap between output and computation. The assembly routines used by the compiler are contained in the IBM 1130 Commercial Subroutine Package, (11) These routines and their uses are given in Table I.

TABLE I

COMMERCIAL SUBROUTINE PACKAGE SUBPROGRAMS
USED ON THE IBM 1130

Name	Description
FILL	Propagates a character through a field.
MOVE	Moves characters from one integer array to another.
PACK	Converts values in an integer array from FORTRAN A1 EBCDIC characters to A2 EBCDIC.
READ	Causes a card to be read on the 1442 card read/punch into an integer array. Characters are represented in A1 EBCDIC.
TYPER	Prints a line of A1 EBCDIC characters on the 1130 console typewriter.
UNPAC	Converts values in an integer array from FORTRAN A2 EBCDIC characters to A1 EBCDIC.

Scratch files, used for temporary storage, presented another minor

problem on the 1130 in that temporary files cannot be formally passed between different phases of the compiler. This problem has been overcome by the way the 1130 Disk Monitor System allocates working storage files. Files are always allocated starting at the same disk address, so by defining scratch files in the same order in each segment of the compiler, the files are allocated in the same place on the disk. Data within the files is not changed during the loading of program phases.

Virtual Memory System

In order to allow the compilation and execution of large ALGOL 68 programs on the 1130, a simple virtual memory system was developed. Two 80-word pages reside in memory during execution. One page is a section of the object program. Since the object program is never modified during execution, no facility for storage into it has been included. Should an attempt be made to store into the program page a storage protection error is indicated. The second page is a segment from the dynamic storage area. It contains identifier storage and program linkage information. When information is stored into this page it is recopied to disk before a new page is swapped in.

All references to the 1130 virtual memory system are made with subroutine calls. It was intended that the virtual memory system be replaced by in-core storage on systems with ample memory. This change can be made easily by intercepting subroutine calls to the fetch and store routines.

As expected, the use of the virtual memory system results in slow execution by the interpretive executor due to disk accesses. It does, however, serve its intended purpose by allowing the definition of non-

trivial ALGOL 68 programs. The performance of the virtual memory system could be improved by implementing a larger page size.

CHAPTER III

THE COMPILER

Compiler Initialization

Setting Compiler Options

The compiler must initially prepare itself for a particular compilation. Not only must internal pointers and data tables be established, but the compiler also must recognize user controlled options to provide various levels of programmer support. These options range from a source listing of the user's program to a compiler dump of intermediate results during compilation.

This particular implementation varies somewhat from the formal definition of the language since compiler options are set by parameters on a control card rather than with pragmats. The control card method was selected because it separated compiler initialization from the scanning and program analysis phases. It also allows for easier keyword identification since options must be specified in a rigid format.

Compiler option keywords are identified by performing a linear search on a keyword table. Although the linear search technique is not particularly efficient, the relatively small list length allows for keyword identification within a reasonable time. A set of default options is provided for the user, so compiler options need be explicitly stated only when special compiler services are desired. The default options

are underlined in the list of compiler options in Table II.

Due to the linear nature of the keyword list, modifications to it are relatively simple. Keywords can be added or deleted without regard to list order. The compiler options which are in effect are indicated by a set of control flags. Simple options, which are either enabled or disabled (e.g., SOURCE/NOSOURCE), require only an array entry to designate which control flag should be set. Options which allow variable parameters (e.g., SORMGIN) require additional program coding to effect changes.

Initializing Common Storage

Parameters are passed between compiler phases through FORTRAN common storage. Since common storage cannot be data initialized it is necessary to assign values at execution time. The initialization phase of the compiler is responsible for setting many of the pointers which are used during the compilation.

The Scanner

Coding Atomic Symbols

To facilitate the analysis of an ALGOL 68 program, atomic symbols are converted to integer values. This eliminates the problems involved with processing symbols of different lengths and allows classification of keywords and operators according to the way in which they are used. Negative integer values are used to denote identifiers while positive integers indicate operators and keywords. Classification of keywords according to use is achieved by assigning equivalent integer values within a specified range (e.g., declaration tokens range from 401 to 499).

TABLE II

LIST OF COMPILER OPTIONS*

Option	Description
SOURCE/NOSOURCE	Prints a listing of the source program
NEST/NONEST	Prints block nesting levels
OPLIST/NOOPLIST	Lists compiler options which are in effect during a particular compilation
ATR/NOATR	Prints an attribute listing for identifiers (not implemented)
XREF/NOXREF	Prints a cross reference listing of identifiers (not implemented)
STMNT/NOSTMNT	Records statement numbers at run time
TRACE/NOTRACE	Prints the number of a statement before it is executed at run time
DUMP/NODUMP	Provides an instruction trace and dump of memory before each instruction is executed
LIST/NOLIST	Lists 4-tuples of the object code as they are generated
SORMGIN=(xx,yy,zz)	Defines the margins of the source record, where xx is the starting column (default 2); yy is the ending column (default 72); and zz is a column containing a standard ANSI carriage control character which is used to control the source listing (no default)
PASSWORD=JCJ!	Dumps intermediate code from the scanner for use by the implementer and system programmers

*Default options are underlined.

Recognizing Atomic Symbols

Keywords and Identifiers. Identifiers and keywords are identified and isolated using a finite state automaton (Figure 1). A regular grammar which is used for the recognition of identifiers is given in Table III. The process is started when an alphabetic character is recognized and continues as long as alphanumeric characters are input. The character string is then truncated or padded with blanks to eight characters. Although a few keywords are longer than eight characters in length, they can be identified uniquely by an eight character fragment. Identifiers may be of any length but they must be unique in the first eight characters.

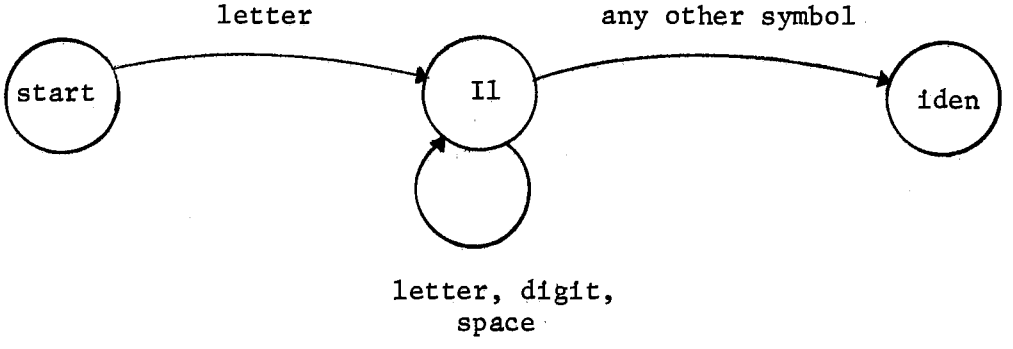

Figure 1. Finite State Automaton for Isolating Identifiers

In order to conserve storage, identifiers are packed two characters per word into four words before they are entered into the symbol table. The symbol table is searched linearly. Coded source program equivalents for identifiers are negative integers.

TABLE III

A GRAMMAR FOR ISOLATING IDENTIFIERS

identifier: tag.

tag: letter; tag, letter; tag, digit token; tag, space.

letter: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.

digit token: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9.

space: _.

Identification of keywords is kept to a minimum during the scan phase. Only keywords concerned with comments (COMMENT and CO) and block structure (BEGIN, END, IF, FI, CASE and ESAC) are identified. All other keywords are treated as identifiers. The keywords are distinguished from the identifiers during the keyword recognition phase.

Operators. Special character operators are identified by means of a table driven finite state automaton. In an effort to save storage, the table is divided into two parts. The first table (Table IV) is used to find a starting state. In some cases, an operator consists of exactly one symbol and the information contained in the first table is sufficient to identify it. Otherwise the second table (Table V) is used as a transition matrix for the finite state automaton. Positive entries in the table indicate a transition is to be made to the specified state. Negative entries indicate an operator has been recognized. Table positions with no entry and characters not contained in the input set signify that the input character is not acceptable and an alternate list must be tested to determine if previously isolated symbols form a valid operator.

TABLE IV

STARTING STATES FOR OPERATOR RECOGNITION

input symbol:	*	/	¬	:	.	-	+	<	>	(	|
start state:	6	8	13	16	21	1	3	14	15	23	24

input symbol:	=	)	&	?	,	@	;	!
coded symbol:	-29	-2	-21	-28	-6	-36	-5	-28

TABLE V

TRANSITION TABLE FOR OPERATOR RECOGNITION

Input Symbol

	*	/	¬	=	:	.	)	
State 1					2			-8 Alt.
2				-7				
3				5	4			-11
4				-9				
5					-10			
6	-13				7			-14
7				-12				
8		9			12		-415	-18
9					10			-19
10				-16	11			-20
11				-17				
12				-15				
13				-22				-23
14				-25				-24
15				-26				-27
16			18	20	17			-35
17				-32				-31
18				19				
19					-34			
20					-33			-30
21				-30		22		-39
22				-30				-35
23		-414						-1
24					-38			-37

An error condition exists when an invalid input symbol is found and there is no entry in the alternate list.

Given the standard set of operators for ALGOL 68, the finite state automaton is sufficient for operator identification with at most one symbol look ahead. If, however, the user is permitted to declare special character operators, this method may fail due to ambiguities in symbol combinations.

Denotations. Denotations, or ALGOL 68 constants, are also recognized by a finite state automaton. The method used is similar to that presented by Smith, et al. (15) for the recognition of denotations. The grammar corresponding to the finite state automaton for parsing denotations is given in Table VI. A two character look ahead is sufficient to distinguish valid denotations, but in some instances it is not sufficient to recover when an error is detected. As denotations are isolated, they are output as object code. Denotations are represented in the coded source program by a denotation indicator followed by the relative address of the denotation in the object code.

String denotations are defined to be a multiple value. Therefore, a descriptor is output with each string denotation. The address of the string denotation in the coded source program is actually that of its descriptor. Although format denotations are not multiple values, they also are output with a descriptor. The descriptor is used only to contain length information for the format denotation. Formats are not parsed by the compiler. They are stored in character form so they may be analyzed at run time.

TABLE VI

A GRAMMAR FOR ISOLATING DENOTATIONS

---

denotation: integral denotation; real denotation; string denotation; format denotation.

integral denotation: digit token; integral denotation, digit token.

real denotation: variable point numeral; floating point numeral.

variable point numeral: integral denotation, fractional part; fractional part.

fractional part: point symbol, integral denotation.

point symbol: ..

floating point numeral: stagnant part, exponent part.

stagnant part: integral denotation; variable point numeral.

exponent part: times ten to the power choice, power of ten.

times ten to the power choice: E.

power of ten: plusminus, integral denotation; integral denotation.

plusminus: +; -.

string denotation: quote symbol, string item sequence, quote symbol.

quote symbol: ".

string item sequence: string item; string item sequence, string item.

string item: ¢ any symbol except the quote symbol ¢; quote symbol, quote symbol.

Keyword Recognition

Keyword Identification

Keyword identification is performed in a phase separate from the scanner. This is due primarily to storage considerations. The list of ALGOL 68 reserved words is lengthy and could not have been included easily with the scanner.

Regardless of how many times a keyword is used in a program, there is only one entry in the symbol table for it. Only one search of the keyword table must be made to identify it throughout the entire program. The keyword search is performed by determining the length of the symbol and using a linear search of a table of fixed length entries. When a keyword is identified an entry is made into an array. The array is then used to map identifier codes to the correct keyword code during a fix-up pass through the coded source program.

The separate pass through the coded source program to identify keywords does not appear to significantly affect the performance of the compiler unless a large number of identifiers and keywords are used within a single ALGOL 68 program. Improved symbol table techniques could be employed to improve performances in these cases. Also, the extra pass through the coded source program allows the identification of labels and an analysis of the block structure of the program.

Label Recognition and Block Analysis

Due to the manner in which branching is controlled in the code generation phase, it is important to know which identifiers are used as labels as well as where they are declared. This requires an extra pass

through the program, but it has been combined with the keyboard recognition pass in this implementation. Labels are identified as an identifier followed by a colon symbol, where they do not occur within a row declaration. When a label is found, a symbol table entry indicating the block in which the label is defined is made.

Program blocks are numbered consecutively by counting open symbols ('(') and their equivalents (BEGIN, IF and CASE). For each block in the program a table entry is made indicating the number of the block which immediately contains it. From this table, and from the block number of a label in the symbol table, the code generation routine can determine if a jump is valid.

Environmental Enquiries

According to the formal definition of ALGOL 68, environmental enquiries and pre-defined constants are identifiers which are declared and initialized in the standard prelude. (18) This implementation handles them in exactly this manner. When an environmental enquiry or pre-defined constant is used within a particular program, code is generated to allocate and initialize it outside of the first block of the user's program.

Parsing and Code Generation

Parsing

Declarations. In an effort to make parsing of declarations simple, declarations have been limited to simple data types and arrays. Expressions are not allowed within a declaration. Furthermore, all identifiers must be declared and declarations may only appear at the beginning of a

block. The grammar used for parsing declarations is given in Table VII.

The restricted declaration format makes it possible to parse declarations by means of a finite state automaton with only one symbol look ahead (Figure 2). This look ahead, however, is not sufficient to recover from error conditions, in which case the remainder of the declaration is ignored. Error states are recognized by invalid input symbols.

Since a run time symbol table is maintained, code must be generated for each symbol declared. The code generated is an instruction which makes the appropriate run time symbol table entry. No address resolution is necessary at compile time since the address is determined from the symbol table at execution time. Identifiers are, in effect, addressed by their symbol (a negative integer).

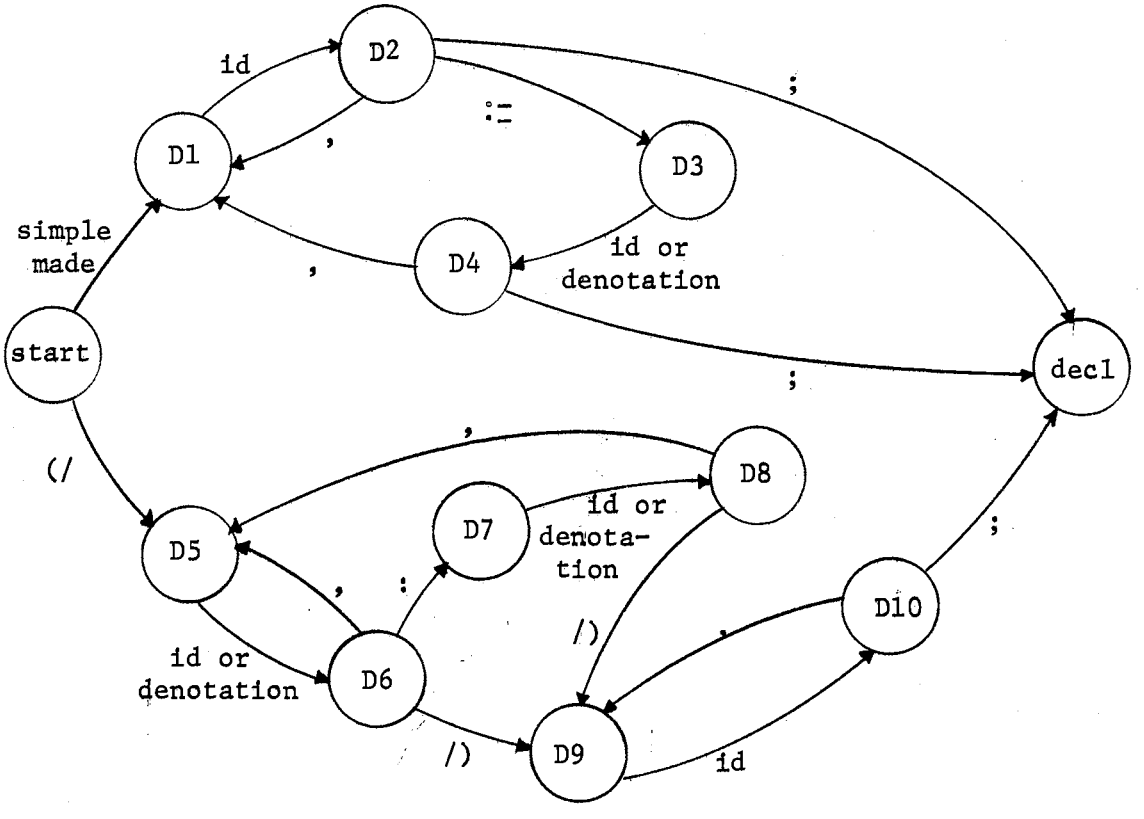

Figure 2. Finite State Automaton for Parsing Single Declarations. The DK's represent arbitrary states in the automaton

TABLE VII

A GRAMMAR FOR DECLARATIONS

declaration prelude sequence: single declaration; declaration prelude sequence, single declaration.

single declaration: simplemode, decl list, semicolon symbol; sub symbol, rows of, bus symbol, simplemode, identifier list, semicolon symbol.

semicolon symbol: ;.

sub symbol: (/.

bus symbol: /).

simplemode: INT; REAL; COMPL; COMPLEX; BOOL; CHAR.

rows of: row of; rows of, comma symbol, row of.

comma symbol: ,.

row of: bound, colon symbol, bound; bound.

colon symbol: :.

bound: integral identifier; integral denotation.

decl list: decl; decl list, comma symbol, decl.

decl: identifier, becomes symbol, identifier; identifier, becomes symbol, denotation; identifier.

becomes symbol: :=.

identifier list: identifier; identifier list, comma symbol, identifier.

Associated with multiple values (rows or arrays) is a descriptor. (18) As a row declaration is parsed, a descriptor template is output with the object code. When the row declaration is elaborated at execution time, the template is used to supply bound information for the actual descriptor.

Program Parse. The ALGOL 68 program is parsed using a combination of top-down and bottom-up methods. The overall program structure, specifically the block structure, is examined from the top down. Individual statements (expressions) of the program are parsed from the bottom up. This method is possible because every identifier, operator and external object (e.g., serial clause) is said to possess a value upon its elaboration. (18) Therefore, it is never necessary to consider anything more complicated than two single values and the effect of an operator on them. Whether a value is from a reference-to-integral-mode-identifier or a strong-conditional-void-clause is insignificant. Table VIII contains a grammar for parsing the body of the ALGOL 68 program.

Expressions are analyzed by means of a modified postfix Polish conversion routine. Output from the routine consists of interpretive code which will cause the evaluation of the expression at run time.

Delimiting symbols (e.g., (, |, ;, ,) present no problem for the parse. They are assigned a priority and are processed in the same manner as other operators. Similarly, procedure calls are treated as monadic operators which are applied to their parameter lists.

Recursive Descent. In order to consider only a single value at a time during the parse, it is necessary to consider each block in the ALGOL 68 program separately. Whenever an open symbol is encountered,

TABLE VIII

A GRAMMAR FOR PROGRAM PARSING

---

particular program: begin symbol, serial clause, end symbol; left parenthesis symbol, serial clause, right parenthesis symbol; if symbol, serial clause, then symbol, unitary clause list, else symbol, unitary clause list, fi symbol; if symbol, serial clause, then symbol, unitary clause list, fi symbol.

serial clause: declaration prelude sequence, unitary clause list, unitary clause list.

unitary clause list: unitary clause; unitary clause list, semicolon symbol, unitary clause.

unitary clause: label, colon symbol, unitary clause; expression; jump.

jump: goto symbol, label; label.

label: identifier.

goto symbol: GO, TO; GOTO.

expression: formula.

formula: formula, p1 operator, p2 operand; p2 operand.

p2 operand: p2 operand, p2 operator, p3 operand; p3 operand.

p3 operand: p3 operand, p3 operator, p4 operand; p4 operand.

p4 operand: p4 operand, p4 operator, p5 operand; p5 operand.

p5 operand: p5 operand, p5 operator, p6 operand; p6 operand.

p6 operand: p6 operand, p6 operator, p7 operand; p7 operand.

p7 operand: p7 operand, p7 operator, p8 operand; p8 operand.

p8 operand: p8 operand, p8 operator, p9 operand; p9 operand.

p9 operand: p9 operand, p9 operator, monadic operand; monadic operand.

monadic operand: monadic operator, monadic operand; primary.

primary: identifier; denotation; selector, primary; slice; particular program; standard procedure, left parenthesis symbol, parameter list, right parenthesis symbol; cast.

TABLE VIII (Continued)

selector: RE; IM.

slice: identifier, sub symbol, indexer list, bus symbol.

indexer list: indexer; indexer list, comma symbol, indexer.

parameter list: unitary clause list; parameter list, comma symbol, unitary clause list.

begin symbol: BEGIN.

end symbol: END.

left parenthesis symbol: (.

right parenthesis symbol:).

if symbol: IF.

fi symbol: FI.

then symbol: THEN.

else symbol: ELSE.

the parse literally starts over with a new program. Only the symbol table reflects the existence of containing blocks. Each block is assumed to contain a complete program which returns a value.

When restarting the parse at the entry to a block, it is necessary to preserve the status of the parse of the containing block. This is done by making entries to the symbol and operator stacks used by the Polish string routine. The operator stack entry contains a priority low enough to prevent the code generation routine from acting on it. along with the block number of the containing block. The symbol stack entry contains information as to where the parse left off. After flags have been reset, the parse starts over. At block exit, the flags are reset from the information which was stacked, and the parse resumes. This

method allows block nesting to occur to an unspecified and theoretically limitless depth.

Code Generation. Each operator is assigned a priority to specify when an operation is to be performed in relation to other operators. The Polish string conversion routine maintains this relationship. Operators and their priorities are passed to this routine and are stacked on the operator stack so that their priorities are strictly non-decreasing. If necessary, the code generation routine pops a higher priority operator from the operator stack, along with any associated operands from the symbol stack, so that this order is preserved. As an operator is popped by the code generation routine, an instruction is generated to perform that operation at run time.

Transfer of Control. Programmer directed jumps and branching associated with conditional clauses also are controlled by the Polish string conversion routine. Jumps, or GOTO's, are indicated by an identifier on the symbol stack with mode label (actually, reference-to-procedure). If the label identifier is defined within a containing block, a branch instruction is generated. Otherwise an error condition is indicated. Since the branch is to a label identifier, no address resolution is necessary during the code generation phase. The address is determined by the parser and entered into the run time symbol table before execution.

Conditional branching is somewhat more complex. As previously stated, the delimiters THEN, ELSE and ELSF are treated as operators. Each generate a branch instruction as they are placed on the operator stack and each carries with it the address of that instruction so that

the branch address can be resolved when code generation for the clause is complete.

When a then-symbol is found, an instruction is generated which causes a jump if the result of the boolean if-clause is false. The branch address is left unresolved until an else-symbol is found. (If there is no else clause, the fi-symbol resolves the address). At that time, an unconditional branch is generated to skip over the else clause, and the branch address for the then-symbol is resolved. When the fi-symbol is found, the unconditional branch address preceeding the else clause is resolved. Using this method, no fix up pass through the object code is needed since all addresses are resolved during the parse. The code generated to control branching in a conditional clause can be seen in Figure 3.

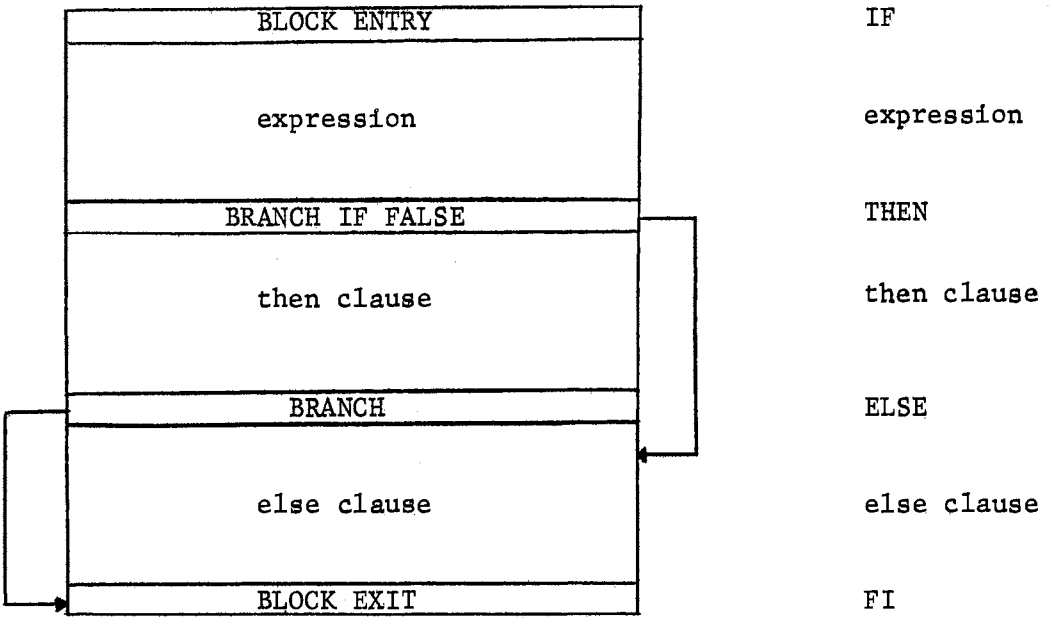

Figure 3. Code Generated for a Conditional Clause

Mode Coercion and Subscripting. Included in the standard prelude for operators are provisions for operations between operands of different modes. Specifically, it allows widening, or coercion from integral to real and from real to complex. Rather than supply each operator in the executor with widening capabilities, an op code was defined to effect widening. This instruction is generated automatically when mode conflicts are noted and widening is appropriate.

Slicing, or subscripting, is accomplished by means of a "load subscripted" instruction which places the address of a single row element into the run time symbol table for a temporary identifier entry. The row element can then be addressed directly through the identifier.

Error Recovery

In an effort to assist the ALGOL 68 programmer, the compiler initiates correctional actions to try to continue the parse. The actions are based on the concept of maintaining a correct relationship between operators on the operator stack and symbols on the symbol stack. Dummy operators and operands are generated as needed for this purpose.

The scanner converts the user's program to a coded form to facilitate program analysis. Parsing of the ALGOL 68 program is performed in two passes using a combination of top down and bottom up methods. Object code consisting of 4-tuples to be interpretively executed is output. Address entries for all labels defined in the ALGOL 68 program are made in the run time symbol table before execution is passed to the executor.

CHAPTER IV

INTERPRETIVE EXECUTION

The Use of FORTRAN

In an effort to make the ALGOL 68 implementation machine independent, IBM 1130 basic FORTRAN was selected for the implementation of interpretive execution. (10) Since FORTRAN is a universal language, code generated by the compiler can be executed on virtually any system through the executor. Also, the object code can be produced by one machine and executed by another.

Using FORTRAN interpretation, it is possible to utilize the functions in the FORTRAN subroutine library. This made implementation on much easier in that existing procedures could be used for computational as well as input/output functions. Although the original implementation uses only standard subroutines, it is possible to modify the executor to include available software. One such change might be to use the complex arithmetic software of a system, rather than depend on the routines in the executor.

The Interpretive Code

Each instruction of interpretive code consists of a 4-tuple of integer values. The first value consists of the operation code and the mode of the instruction. The use of the remaining three values is dependent upon the instruction. They may contain addresses of up to three

operands, or additional mode or operation information. The operation codes are listed in Appendix B.

Storage Management

The execution time layout of storage is shown in Figure 4. The object program resides in the low end of the storage area. This area is fixed and is not modified during program execution. The remaining storage is used to contain two stacks. Dynamic storage for identifiers and system linkage needs is allocated from the execution stack. Heap storage, for identifiers with the HEAP attribute, is allocated in a stack-like manner from the other end of storage.

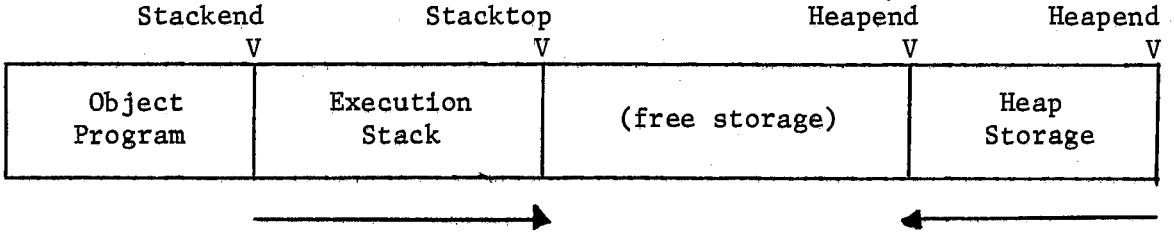

Figure 4. Storage Layout

The limits of storage are set by the stackend and heapend pointers. The heapend pointer is set to reflect the absolute limit of the storage area and the stackend pointer indicates the end of the program area. The limits of allocation are also maintained by pointers. The stacktop pointer is incremented as storage is allocated so that it points to the next available storage location. Similarly, the heaptop pointer is decremented as heap storage is allocated. Stack overflow occurs when the stacktop pointer is greater than the heaptop pointer, but this occurs

only when every available storage location has been used.

The storage area for the executor is addressed in three different ways. Addresses which refer to a constant value or a fixed branch address are absolute, relative to the beginning of the storage area. These addresses are represented in the operand fields of the interpretive instructions as positive values. References to values possessed by identifiers are indirect and must be resolved from the run time symbol table. A reference to an identifier is indicated by a negative operand value which refers to a relative position in the symbol table. Temporary storage for intermediate results is allocated as needed from the top of the execution stack and is referenced by an operand with a value of zero. Addresses for values on the top of the stack are computed from the stack-top pointer and the mode information contained within an instruction. Stack temporaries are automatically allocated or released as they are referenced. Regardless of the type of addressing which is used, a subroutine call is made to determine the absolute address of an operand.

DISPLAYS

The block structure of a program is maintained through the use of DISPLAYS. (6) The DISPLAY provides a convenient mechanism for maintaining system linkages when storage is allocated in a stack-like manner. Contained within the DISPLAY are pointers to DISPLAYS for all containing blocks as well as a pointer to the head of a linked list of identifiers declared within the block.

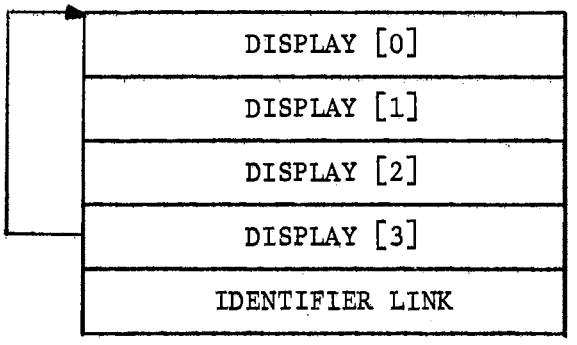

Figure 5. DISPLAY for a Nesting Level of Three

The pointers to higher level DISPLAYS provide a means by which no storage need be explicitly released at a block exit. (17) The entry pointing to the current DISPLAY is actually the stacktop pointer before the most recent block entry was made. Restoring the old stacktop pointer automatically releases storage allocated by the block. Similarly, multiple block exists, needed when control is transferred to an outer block, can be effected by restoring the appropriate stacktop pointer (Figure 6). This method for multiple exists is not used in this implementation, however, because the run time symbol table must be restored at the exit from each block.

Run Time Symbol Table

Declaring Identifiers

As a declaration is elaborated, storage is allocated for the identifier and the run time symbol table is updated to reflect the new declaration. Previous declarations are preserved by storing the symbol table entries on the execution stack in a linked list. The DISPLAY for each

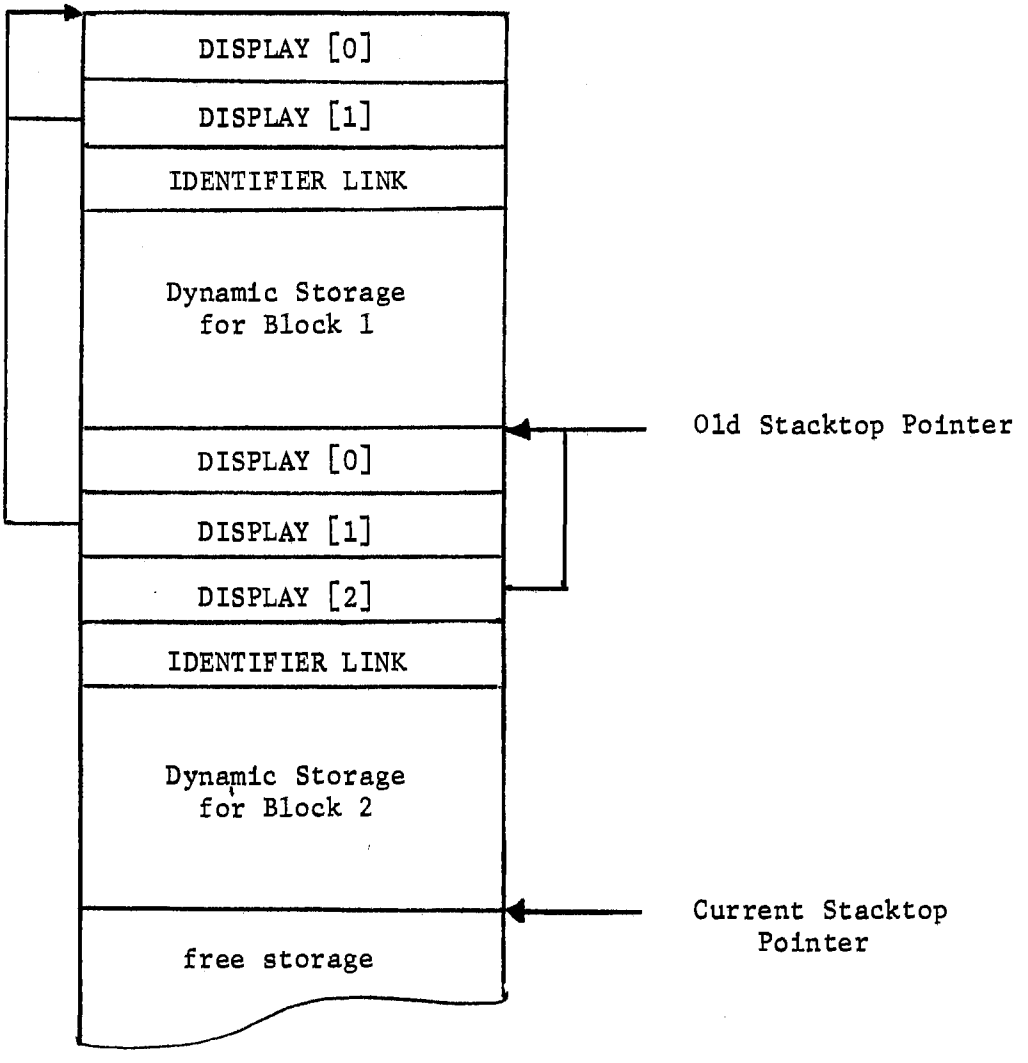

Figure 6. Relation of Stacktop Pointers to DISPLAY Entries

block contains a pointer to the head of the linked list for all identifiers declared within the block (Figure 7). At block exit, the symbol table is restored from the previous symbol table entries in the linked list.

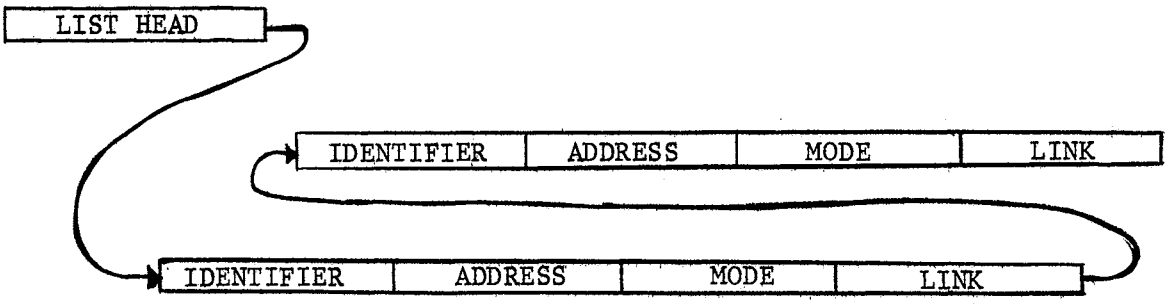

Figure 7. Chaining of Previous Declarations

The use of the run time symbol table eliminates the problems involving the reach of a declaration. Only the most recent declaration is available from the symbol table. Previous declarations are not available until the block containing new declarations is exited. Also, identifiers which have not been declared before they are referenced can be detected easily by invalid addresses in the run time symbol table.

Subscripting

Reference to a single element of a multiple value is effected through a special symbol table entry. Identifiers are generated by the compiler to contain the address of the element. An instruction is used to compute the address using information contained in the multiple value descriptor (Figure 8) and the subscripts which are in temporary storage on top of the execution stack.

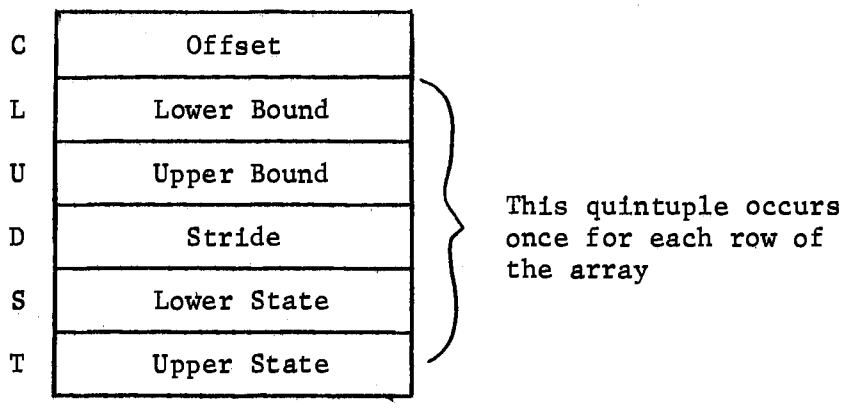

Figure 8. Format of the Multiple Value Descriptor

Subscript checking occurs automatically during interpretive execution. Subscript bound information is obtained from the descriptor for the multiple value. Upon recognition of a subscripting error, a message is printed and execution is terminated. If all subscripts are within the correct bounds, the address is computed as an offset from the first element in the array and the address is stored into the symbol table under an entry for a special identifier.

Error Processing

Run time error checking provides a mechanism for detecting errors which can not be detected easily by the compiler. This is limited primarily to operations which are not defined between arguments of certain modes. These errors are detected automatically during instruction of code decoding and execution is terminated.

Other run time error checking involves testing for identifiers which have not been declared. This is accomplished through the use of

the run time symbol table. Mode checking is not performed at execution time since all mode conflicts are resolved by the compiler.

Execution error messages are printed by a separate abnormal termination routine. The correct error message is determined from the completion code which is set by the executor. As a user specified option, statement numbers are retained during execution to make error analysis easier. The abnormal termination routine also dumps internal pointers when debugging aids are enabled. This facilitates detection of system errors.

The 4-tuples of code generated by the compiler are interpretively executed. Dynamic storage is administered in a stack-like manner by the executor. A run time symbol table is maintained for address resolution of identifiers.

CHAPTER V

USER'S GUIDE

Control Cards

The control cards necessary for execution of the ALGOL 68 subset compiler on the IBM 1130 are shown in Figure 9.

```
// JOB
// XEQ ALG68
:JOB
    .
    .
    ALGOL 68 Source Program
    .
    .
:ENTRY
    .
    .
    Program Data (if any)
    .
    .
:IBSYS
```

Figure 9. ALGOL 68 Control Cards for the IBM 1130

The :JOB card initiates program input. Compiler options (Table II) may be specified using the format specified in Figure 10. A blank terminates the scan of the :JOB card and the remainder of the card is treated as comments. Table IX contains a list of abbreviations which may be used to specify compiler options. The word NO may be used in front of

abbreviations as well as keywords, where appropriate, to turn off unwanted options.

columns 1-4	the characters :JOB
columns 6-22	accounting information (not used)
columns 23-	compiler options, each preceeded by a comma and containing no embedded blanks

Figure 10. Format of the :JOB Card

TABLE IX

COMPILER OPTION ABBREVIATIONS

Abbreviation	Keyword
S	SOURCE
N	NEST
OP	OPLIST
A	ATR*
X	XREF*
ST	STMNT
T	TRACE
D	DUMP
L	LIST
SM	SORMGIN
PW	PASSWORD

*ATR/XREF have not been implemented.

Restrictions

The following restrictions have been applied to the ALGOL 68 subset:

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;

4. All identifiers must be declared before they are referenced. All declarations must appear at the front of a block;

5. Identifiers may not contain embedded blanks, but the break character or underline (_) may be used to improve readability;

6. Comments may appear anywhere a blank may appear, but they must start and end with the same symbol;

7. A label may be defined only once within a program and may not subsequently be declared as an identifier;

8. Conditional clauses must be balanced by the programmer;

9. Identifier initializations may contain no expressions and must be of the correct mode;

10. Bounds on array declarations must be simple integral denotations or identifiers.

Programming Tips

The user should avoid the use of unnecessary parenthesis and BEGIN-END blocks. A considerable amount of overhead is involved with block entry and exit.

Subscripting for multiply dimensioned arrays is significantly slower than for singly dimensioned arrays. Where possible, the number of dimensions should be kept to a minimum.

When the 1130's virtual memory system is in use, multi-dimensional arrays should be initialized in row order to minimize the number of page faults.

CHAPTER VI

SYSTEM PROGRAMMER'S GUIDE

Compiler Modification

Changing Symbol Table Sizes

In all phases of the compiler, as well as the interpretive executor, the symbol table is the last entry in the COMMON storage area. The symbol table size can be modified by changing the array dimension for SYMTB. (In the forth phase of the compiler, STACK is EQUIVALENCEd to SYMTB so its dimension also should be changed.) References to the length of the symbol table are made by a reference to the integer variable SLEN, so its DATA initialized value must be changed to correspond to the symbol table dimension.

The parsing and code generation phase of the compiler indirectly refers to the run time symbol table length of the executor through the integer variable MAXSM. This value represents the maximum number of symbols which can be entered into the symbol table during program execution. Since symbol table entries require two words, MAXSM has a value equal to one half of the run time symbol table length.

Converting to In-core Storage

On systems with sufficient core storage, the 1130 virtual memory system should be eliminated to increase the speed of execution. A single

region of storage is required for the compiler's memory needs and ideally it should be addressable from all routines. This could be achieved using labeled COMMON or by appending the additional storage on the end of the system's blank COMMON.

Fetching and storing into memory is performed by the subroutines ALGFE and ALGST, respectively. These routines would have to be rewritten to allow the direct reference to storage. The coding of these routines for use with storage in labeled COMMON is shown in Figure 11.

```
      SUBROUTINE ALGFE(IADDR,BUFFER,LENG)
C     FETCH ROUTINE
      INTEGER BUFFER(1)
      COMMON/MEMRY/ M(8000)
      K=IADDR+LENG-1
      J=1
      DO 10 I=IADDR,K
      BUFFER(J)=M(I)
   10 J=J+1
      RETURN
      END

      SUBROUTINE ALGST(IADDR,BUFFER,LENG)
C     STORE ROUTINE
      INTEGER BUFFER(1)
      COMMON/MEMRY/ M(8000)
      K=IADDR+LENG-1
      J=1
      DO 10 I=IADDR,K
      M(I)=BUFFER(J)
   10 J=J+1
      RETURN
      END
```

Figure 11. Fetch and Store Routines for Conversion to Core Storage

Converting I/O Routines

The routines used on the 1130 to perform input and output are, by

necessity, system dependent. Therefore, calls to these routines are made in such a manner that I/O routines could be written easily for any system to perform these functions. All input and output is done in terms of A1 EBCDIC character strings stored in integer arrays. The compiler does all of its own formatting and data conversion. Suggested routines to convert to standard I/O are in Figure 12.

```
      SUBROUTINE ALGPR(LINE,LENG)
C     PRINT ROUTINE
      INTEGER LINE(1)
      DATA IOUT/..../
      WRITE(IOUT,10)(LINE(I),I=1,LENG)
   10 FORMAT(120A1)
      RETURN
      END

      SUBROUTINE READ(CARD,START,STOP,EOF)
C     READ ROUTINE
      INTEGER CARD(80),START,STOP,EOF
      DATA IN/..../
      READ(IN,10,END=30)(CARD(I),I=START,STOP)
   10 FORMAT(80A1)
   20 RETURN
C     END-OF-FILE
   30 EOF=1
      GO TO 20
      END
```

Figure 12. Read and Print Routines for Conversion to Standard I/O

Card input involves reading characters from a card and storing them into specified positions of an integer array. This function is performed by subroutine READ. This subroutine contains a parameter to indicate a last card, or end-of-file, sequence on the 1442 read/punch, but this parameter is not actually used by the compiler.

Output is generated one line at a time and is passed to subroutine

ALGPR to be printed. The print line is a variable length integer array of FORTRAN A1 EBCDIC characters, the first of which is for carriage control. The carriage control characters are not printed and are standard ANSI carriage control characters as shown in Table X.

TABLE X

CARRIAGE CONTROL CHARACTERS

Character	Action
(blank)	single space
0	double space
-	triple space
+	no advance
1	new page

Program Linkage

The IBM 1130 implementation consists of distinct program segments which are loaded and executed as needed. It is never necessary to have any portion of a previous phase in core during the execution of a subsequent phase. (Some subroutines are used in more than one phase, but each core image program contains its own copy of such routines.)

The loading of compiler phases is program controlled on the 1130 by the system LINK instruction. (12) This instruction signals the core load builder to load and begin execution of a specified program, namely the next phase of the compiler. Parameters in COMMON storage are preserved during the LINK operation.

When converting to other systems, overlay capabilities of a linkage editor could be used to achieve the effect of the 1130 LINK instruction. Each segment of the compiler could be overlayed when it is no longer

needed. Provisions would have to be made concerning subroutines which are shared by different program segments. This could involve keeping them resident in core of by supplying a new copy with each overlay. Values in COMMON storage must remain unchanged during program segment overlaying.

Figure 13 suggests a method by which use of overlays could be avoided. This would involve changing the parameters of the 1130 FORTRAN CALL LINK instructions from program names to integer values. Program segments would be treated as subroutines whose execution is controlled by a subroutine call 'LINK'. Subroutine LINK is called when a transfer of control to the next phase is desired. It should be noted that this linkage method results in subroutine LINK being called recursively, thus possibly destroying system linkage information. However, depending on the operating system being used, this information may never be used, since a normal return would never be executed. Program control is never returned to a previous phase of the compiler. When the compiler or executor has finished, a CALL EXIT command is issued to return to the operating system. If an operating system can regain control without using the previously destroyed linkage information, this method could possibly be used.

Data Alignment and Lengths

All numeric and character values, regardless of length, are treated as elements of integer arrays. This requires the ability to redefine different values as integer arrays through the use of the EQUIVALENCE statement. The manner in which this is done is highly machine dependent due to system differences in storage and length requirements for each

data type.

1) Replace program names in CALL LINK instructions with the following integer values:

Program name	Integer value	Program function
ALG02	1	scanner
ALG03	2	keyword recognition
ALG04	3	parsing and code generation
ALG05	4	interpretive executor
ALGAE	5	abnormal termination handling
ALGCD	6	compiler debugging
ALGCE	7	compiler debugging

2) Change all of the above mainline routines to subroutines

3) Include the following program to control linkage:

```
      SUBROUTINE LINK(N)
      GO TO(10,20,30,40,50,60,70),N
10    CALL ALG02
20    CALL ALG03
30    CALL ALG04
40    CALL ALG05
50    CALL ALGAE
60    CALL ALGCD
70    CALL ALGCE
      END
```

Figure 13. Steps in Conversion to Linkage by Subroutine Call

The length for each data type on a particular system is indicated by an entry in the integer array LENGS. This array contains five entries which indicate the length in words of integral, real, complex, boolean and character values, respectively. It should be noted that boolean and character values are treated as integer values, while complex values are represented as a pair of real values.

Implementing Procedures

Parsing Procedure Declarations

Since procedure declarations may contain expressions, procedures must be parsed by the routine used for analyzing unitary clauses (ALG04). The declaration parsing routine (ALGDL) is simply not sophisticated enough to handle the types of statements which could occur in procedures. A suggester grammar for parsing procedure declarations is given in Table XI.

TABLE XI

A GRAMMAR FOR PROCEDURE DECLARATIONS

procedure declaration: proc symbol, identifier, formal parameter pack, proc mode, colon symbol, unitary clause, semicolon symbol; proc symbol, identifier, proc mode, colon symbol, unitary clause, semicolon symbol.

formal parameter pack: left parenthesis symbol, formal parameter list, right parenthesis symbol.

formal parameter list: virtual parameter; formal parameter list, comma symbol, virtual parameter.

virtual parameter: parameter mode, identifier list.

parameter mode: proc rows, simplemode; simplemode; reference-to symbol, proc rows, simplemode; reference-to symbol, simplemode.

proc rows: sub symbol, rows, bus symbol; sub symbol, bus symbol.

rows: comma symbol; rows, comma symbol.

proc mode: simplemode; VOID.

proc symbol: PROC.

reference-to symbol: REF.

It is suggested that procedure declarations be restricted to allow no reference to identifiers which have not been previously declared by the user, either in the parameter list or in a containing block of the ALGOL 68 program. The mode of all operands must be known before code generation can occur. Therefore, if an identifier has not been declared, a valid instruction referencing it cannot be generated. An alternative to this restriction would be to make an extra pass through the coded source program to determine the modes of all identifiers in the program.

Additional Instructions

At least four additional instructions must be defined to effect procedure calls: an instruction for passing parameters; an instruction to link to a procedure; an instruction to return from a procedure; and a branch instruction for abnormal exit from a procedure.

The passing of parameters could be handled easily through the run time symbol table. All that would be involved would be to place the addresses of the actual parameters in the calling statement into the symbol table entries for the virtual parameters of the procedure declaration. Since address resolution occurs at run time, there is no problem as to which allocation of an identifier would be referenced. Only the most recent declarations would be available from the symbol table.

Linking to a procedure is relatively simple. It involves storing the return address on top of the execution stack and branching to the first instruction of the procedure. At the procedure entry point, parameter addresses, also stored on the execution stack, would be retrieved. Exit from the procedure would involve restoring the symbol table and branching to the return address.

The use of DISPLAYS makes abnormal exits from procedures easy. Since the nesting level of the block for each label is known, storage release and transfer of control are greatly simplified. It is, however, necessary to restore the run time symbol table to account for all declarations elaborated within blocks which are exited.

Program Control Flag

An extra control flag is necessary to indicate that a procedure declaration is being parsed. This flag would be tested each time a semicolon is found to determine if it ends a procedure declaration. If so, an exit instruction would be generated to return to the calling statement. This flag also could be used to signal that a parameter list is being parsed, in which case an address, rather than a value, would be loaded on the execution stack when a comma is found. Then, when a right parenthesis is found the procedure call would be generated. This program control flag would be stacked as is the block number during recursive descent.

Error Checking

Detection of errors concerning procedure calls could become quite complex. Minimally, a check must be made to see if the number of parameters in a call matches the number in the procedure declaration. This would be sufficient to prevent a system error but would require that the user be responsible for mode checking. Ideally, the parser should generate tables for mode checking of procedure operands. This would eliminate user errors involving referencing parameters.

Compiler Debugging Aids

Within the compiler are debugging facilities so the compiler and executor can help to debug themselves. This allows system programmers and the implementor to dump intermediate values between phases of the compiler. All values dumped are clearly labeled and indicate the status of the compilation at various points.

The compiler debugging facility is not intended for use by the application programmer. It does not provide him with any useful information for debugging and ALGOL 68 program.

The debugging aids are enabled through the PASSWORD=JCJ! parameter of the :JOB card. Also of interest to the system programmer are the LIST and DUMP options. The LIST option prints 4-tuples of generated code (some branch addresses will not be resolved). The DUMP option provides an execution trace, along with a complete dump of the dynamic storage stack, before each instruction is executed. Use of the DUMP option should be limited since potentially large amounts of output are possible.

CHAPTER VII

SUMMARY, CONCLUSIONS AND FUTURE WORK

Using the methods outlined in this paper an ALGOL 68 subset has been implemented at Oklahoma State University for the IBM 1130. It has successfully compiled and executed programs of a scientific nature. To a limited extent the translator has been used by undergraduate students in an attempt to learn the language.

Future work on the system will involve extensive testing and the implementation of extended transput capabilities. Additional compile time error checking would be desirable for instructional purposes.

This implementation seems to meet the basic needs for introducing the student to the ALGOL 68 language while it allows compilation and execution of application programs of a scientific nature. This subset translator provides a suitable starting point for an understanding of the complete ALGOL 68 language.

A SELECTED BIBLIOGRAPHY

(1) Berry, Roger. "A BNF Grammar for Formats in ALGOL 68," Proceedings of an ALGOL 68 Workshop, Oklahoma State University, 1973.

(2) Berry, Roger. "A Practical Algorithm for the Implementation of Formats in ALGOL 68," Proceedings of an ALGOL 68 Workshop, Oklahoma State University, 1973.

(3) Berry, Roger. "Transput in ALGOL 68," Proceedings of an ALGOL 68 Workshop, Oklahoma State University, 1973.

(4) Currie, I. F., Susan G. Bond and J. D. Morrison. "ALGOL 68-R," ALGOL 68 Implementation, ed. J. E. L. Peck. Proceedings of the IFIP Working Conference on ALGOL 68 Implementation. Munich, July, 1970. Amsterdam: North-Holland Publishing Company, 1971.

(5) Currie, I. F. and P. M. Woodward. "Working Description of ALGOL 68-R," Royal Radar Establishment Memorandum, Malvern, England, December, 1970.

(6) Gries, David. Compiler Construction for Digital Computers. New York: John Wiley and Sons, Inc., 1971.

(7) Hedrick, G. E. and B. R. Alexander. "A Transition from PL/I to ALGOL 68," Proceedings of the Second Vancouver Conference on ALGOL 68 Implementation, Vancouver, 1972.

(8) Hedrick, G. E. and C. L. Smith. "Context Conditions in ALGOL 68," Soken Kiyo, 3.1 (1973), pp. 1-28.

(9) Hill, U., H. Scheidig and H. Woessner. "An ALGOL 68 Compiler," Technical Report, Technical University of Munich and University of British Columbia, 1972.

(10) IBM 1130/1800 Basic FORTRAN IV Language (GC26-3718).

(11) IBM 1130 Commercial Subroutine Package (1130-SE-25X), Version 3 Mod. 1-Program Reference Manual (GH20-0241).

(12) IBM 1130 Disk Monitor System, Version 2, Programmer's and Operator's Guide (GC26-3717).

(13) Jensen, John C. "An ALGOL 68 Subset Compiler," Proceedings of an ALGOL 68 Workshop, Oklahoma State University, 1973.

(14) Jensen, John C. "Considerations for the Implementation of an ALGOL 68 Subset on the IBM 1130," Proceedings of an ALGOL 68 Workshop, Oklahoma State University, 1973.

(15) Smith, C. L., G. E. Hedrick and J. R. Van Doren. "A Scanning Algorithm for ALGOL 68 Denotations," Proceedings of the Second Vancouver Conference on ALGOL 68 Implementation, Vancouver, 1972.

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

(17) Van Doren, James R. "A Conceptual Model for Dynamic Storage Administration in Block Structured Languages," Technical Note, Oklahoma State University, 1972.

(18) van Wijngaarden, A., ed., B. J. Mailloux, J. E. L. Peck and C. H. A. Koster, "Report on the Algorithmic Language ALGOL 68," Numerische Mathematik, 14 (1969), pp. 457-519.

(19) Woodward, P. M. "Primer of ALGOL 68-R," Royal Radar Establishment Memorandum, Malvern, England, 1970.

(20) Woodward, P. M. and Susan G. Bond. User's Guide to ALGOL 68-R. Malvern, England: Royal Radar Establishment, 1971.

APPENDIX A

CODED PROGRAM EQUIVALENTS

TABLE XII

CODED PROGRAM EQUIVALENTS

Action Tokens

Description	Keyword	Symbol	Equivalent
minus and becomes symbol	MINUSAB	-:=	7
minus symbol		-	8
plus and becomes symbol	PLUSAB	+:=	9
prus and becomes symbol	PRUS	+=:	10
plus symbol		+	11
times and becomes symbol	TIMES	*:=	12
up symbol	UP	**	13
times symbol		*	14
divide and becomes symbol	DIV	/:=	15
over and becomes symbol	OVERB	//:=	16
modulo and becomes symbol	MODB	//::=	17
divide symbol		/	18
over symbol	OVER	//	19
modulo symbol	MOD	//:	20
and symbol	AND	&	21
not equal symbol	NE	¬=	22
not symbol	NOT	¬	23
less than symbol	LT	<	24
less than or equal symbol	LE	<=	25
greater than or equal symbol	GE	>=	26
greater than symbol	GT	>	27
plus i times symbol		!	28
plus i times symbol		?	28
equals symbol	EQ	=	29
becomes symbol		:=	30
becomes symbol		.=	30
becomes symbol		..=	30
conforms to symbol	CT	::	31
conforms to and becomes	CTAB	::=	32
is symbol	IS	:=:	33
isnt symbol	ISNT	:¬=:	34
or symbol	OR		35

Syntactic and Sequencing Tokens

Description	Keyword	Symbol	Equivalent
begin symbol	BEGIN	(	1
case symbol	CASE	(	1
if symbol	IF	(	1
end symbol	END	)	2
esac symbol	ESAC	)	2
fi symbol	FI	)	2

TABLE XII (Continued)

Description	Keyword	Symbol	Equivalent
semicolon symbol		;	5
comma symbol		,	6
colon symbol		:	35
colon symbol		..	35
at symbol	AT	@	36
else symbol	ELSE	|	37
out symbol	OUT	|	37
then symbol	THEN	|	37
in symbol	IN	|	37
else if symbol	ELSF	|:	38
completion symbol		.	39
sub symbol		(/	414
bus symbol		/)	415

Environmental Enquiries and Pre-defined Constants

Name	Equivalent
BITSLENGTH	201
BITSWIDTH	202
BYTESLENGTH	203
BYTESWIDTH	204
INTLENGTHS	205
FALSE	206
NMAXFACT	207
NULLCHAR	208
REALLENGTHS	209
TRUE	210
MAXINT	211
NIL	212
MAXREAL	213
SMALLREAL	214
PI	215

Standard Procedures

Name	Equivalent
CMPLXSQRT	301
NEXTRANDOM	302
ARCCOS	303
ARCSIN	304
ARCTAN	305
CLEAR	306
ENTIER	307
LOG10	308
MATCH	309

TABLE XII (Continued)

Name	Equivalent
PRINT	310
RANDOM	311
ROUND	312
ABS	313
ARG	314
BIN	315
CONJ	316
COS	317
ELEM	318
EXP	319
GET	320
INF	321
LOG2	322
LWB	323
LWS	324
ODD	325
OUTF	326
PUT	327
READ	328
REPR	329
SET	330
SIGN	331
SIN	332
SQRT	333
TAN	334
UPB	335
UPS	336
IM	337
LN	338
RE	339

Declaration Tokens

Name	Equivalent
COMPLEX	401
COMPL	401
EITHER	402
FORMAT	403
STRING	404
BOOL	405
CHAR	406
FLEX	407
HEAP	408
INT	409
PROC	410
REAL	411
REF	412
VOID	413

TABLE XII (Continued)

Special Tokens	
Name	Equivalent
PRAGMAT	601
PR	601
WHILE	602
FOR	603
FROM	604
GOTO	605
SKIP	606
BY	607
DO	608
GO	609
OF	610
TO	612
STANDIN	613
STANDOUT	614
EXIT	615

APPENDIX B

INTERPRETIVE OPERATION CODES

APPENDIX B

INTERPRETIVE OPERATION CODES

Introduction

All interpretive instructions are composed of four integer values. The formats of the instructions vary with different operations but contained within each instruction is the operation code, the operands, the modes of the operands and the location for the result. In some operations, the low order digit of the first operand is used to specify mode information. For these instructions, the two values in the first operand are parenthesized. Table XIII is a key for the values of the mode indicator. Table XIV explains the meanings of most of the symbols which describe the instructions.

TABLE XIII

MODE INDICATOR VALUES

Value	Mode
1	integral
2	real
3	complex
4	boolean
5	character

TABLE XIV

KEY TO INSTRUCTION FORMAT SYMBOLS

Symbol	Description
arg1,arg2	the symbolic addresses for arguments
arg3	the symbolic address of the location into which the result is to be stored
array identifier	the symbolic address of the descriptor for an array
mode	the mode of an operation
op	the operation code
rows	the number of rows in an array

System Instructions

Block Entry

General format: 010,0,0,0

The linkage operations necessary for a block entry are performed by the executor.

Block Exit

General format: 020,mode,0,0

The linkage operations necessary for a block exit are performed by the executor. The mode is the mode of an operand which is to be returned on the execution stack. If no value is to be returned, mode is zero.

Jump

General format: 030,branch code,arg1,arg3

If the branch code is one, an unconditional branch to the address of arg3 is executed. If the branch code is two, the boolean value specified by arg1 is examined. If the value is false, the branch is taken. Otherwise, normal instruction processing continues with the next instruction.

Allocate Symbol

General format: 040,mode,0,identifier

Storage for the specified identifier is allocated from the execution stack and the run time symbol table is updated to reflect the current allocation.

Update Statement Number

General format: 050,stmnt,0,0

The statement number indicator is replaced by the value of 'stmnt.'

Print

General format: 060,mode,0,arg1

The value of the operand is output on the standard output device.

Becomes

General format: 070,mode,arg1,arg3

The value of arg1 is stored into arg3. The value is also stored on top of the execution stack for further use.

Dyadic Operators

General format: (op,mode),arg1,arg2,arg3

The operation is performed between arg1 and arg2 and the result is stored in arg3. Table XV lists the set of dyadic op codes.

TABLE XV

DYADIC OPERATION CODES

OP	Operation
10	add
11	subtract
12	divide
13	multiply
14	exponentiation
15	modulo
16	plus and becomes
17	prus
18	minus and becomes
19	divide and becomes
20	multiply and becomes
21	modulo and becomes
22	not equal
23	less than
24	less than or equal
25	greater than or equal
26	greater than
27	equal
28	and
29	or

Monadic Operators

Simple Monadic Operators

General format: (30,mode),op,arg1,arg3

Simple monadic operations are monadic operations and standard procedure calls in which the mode of the result is the same as the mode of the argument. The operation is applied to arg1 and the result is stored into arg3. Table XVI lists the simple monadic operator operation codes.

TABLE XVI

MONADIC OPERATION CODES

Op	Operation
01	unary plus
02	unary minus
03	absolute value
04	square root
05	exponential function
06	natural logarithm
07	base two logarithm
08	base ten logarithm
09	sine function
10	cosine function
11	tangent function
12	arcsine function
13	arccosine function
14	arctangent function
15	conjugate value

Complex Argument, Real Result

General format: 313,op,arg1,arg3

The operation is applied to the complex argument arg1 and the real result is stored into arg3. Operation codes are given in Table XVII.

TABLE XVII

COMPLEX TO REAL MONADIC
OPERATION CODES

Op	Operation
01	complex absolute value
02	argument function
03	real part of a complex value
04	imaginary part of a complex value

Real Argument, Integral Result

General format: 322,op,arg1,arg3

The operation is applied to the real argument arg1 and the integral result is stored into arg3. Operation codes are given in Table XVIII.

TABLE XVIII

REAL TO INTEGRAL MONADIC OPERATION CODES

Op	Operation
01	entier (floor function)
02	lower bound (floor function)
03	round
04	sign
05	upper bound (ceiling function)

Integral Argument, Boolean Result

Odd Function.

General format: 331,1,arg1,arg3

If the value of the integral argument arg1 is odd, a value of true is stored into arg3. Otherwise, a value of false is stored into arg3.

No Argument, Real Result

RANDOM.

General format: 342,1,0,arg3

A pseudo-random number between 0.0 and 1.0 is generated and stored into arg3.

Character Argument, Integral Result

Character Absolute Value.

General format: 344,1,arg1,arg3

The position in the collating sequence of the character value of arg1 is stored into arg3.

Integral Argument, Character Result

REPR Function.

General format: 351,1,arg1,arg3

The integral value of arg1 is used to index the string of characters in the collating sequence. The character value for the selected position is stored into arg3.

Row Operations

Allocate Descriptor

General format: (50,mode),array identifier, rows, template

A descriptor is created for the identifier specified using information stored in the descriptor template. The array is then allocated and the run time symbol table is updated to reflect the new allocation.

Load Subscripted

General format: 510,rows,array identifier,arg3

The address of a single element of the specified array is placed into the symbol table entry for arg3. Index values are on top of the execution stack.

Coercion

General format: (61,mode),input mode,output mode,arg1

The value of arg1 is widened from the input mode to the output mode and is stored on the execution stack. If 'mode' is nonzero, the value on the top of the stack is saved before coercion occurs and is restored afterwards.

APPENDIX C

THE ALGOL 68 SUBSET

APPENDIX C

THE ALGOL 68 SUBSET

Description

A grammar for the scientific subset of ALGOL 68 is given in Table XIX. The subset is intended to reflect the block structure of the language as well as many of the capabilities for arithmetic expressions.

Although comments are not defined by the subset grammar, they are permitted to allow program documentation. Comments may appear anywhere a blank may appear except within a string denotation. The representations for the comment symbol are ¢, #, COMMENT and CO. A comment must begin and end with the same representation of the comment symbol.

The block structure of an ALGOL 68 program is specified by closes clauses (BEGIN-END blocks) and conditional clauses (IF-FI and CASE-ESAC). The extension allowing BEGIN, IF and CASE to be replaced by a left parenthesis symbol, and END, FI and ESAC by a right parenthesis symbol is included in this implementation. Similarly, THEN, ELSE, IN and OUT may be represented symbolicly by a vertical bar (|). The ELSF (or |:) symbol is permitted as an abbreviation for the symbols ELSE IF. No FI symbol is required after an ELSF clause. Some examples of program blocks are given in Table XX.

TABLE XIX

THE ALGOL 68 SUBSET GRAMMAR

---

particular program: begin symbol, serial clause, end symbol; left parenthesis symbol, serial clause, right parenthesis symbol; if symbol, serial clause, then symbol, unitary clause list, else symbol, unitary clause list, fi symbol; if symbol, serial clause, then symbol, unitary clause list, fi symbol.

serial clause: declaration prelude sequence, unitary clause list; unitary clause list.

declaration prelude sequence: single declaration; declaration prelude sequence, single declaration.

single declaration: simplemode, decl list, semicolon symbol; sub symbol, rows of, bus symbol, simplemode, identifier list, semicolon symbol.

simplemode: INT; REAL; COMPL; COMPLEX; BOOL; CHAR.

rows of: row of; rows of, comma symbol, row of.

comma symbol: ,.

row of: bound, colon symbol, bound; bound.

bound: integral identifier; integral denotation.

decl list: decl; decl list, comma symbol, decl.

decl: identifier, becomes symbol, identifier; identifier, becomes symbol, denotation; identifier.

becomes symbol: :=.

identifier list: identifier; identifier list, comma symbol, identifier.

identifier: tag.

tag: letter; tag, letter; tag, digit token; tag, space.

letter: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.

digit token: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9.

space: _.

denotation: integral denotation; real denotation; string denotation; format denotation.

TABLE XIX (Continued)

integral denotation: digit token; integral denotation, digit token.

real denotation: variable point numeral; floating point numeral.

variable point numeral: integral denotation, fractional part; fractional part.

fractional part: point symbol, integral denotation.

point symbol: ..

floating point numeral: stagnant part, exponent part.

stagnant part: integral denotation; variable point numeral.

exponent part: times ten to the power choice, power of ten.

times ten to the power choice: E.

power of ten: plusminus, integral denotation; integral denotation.

plusminus: +; -.

string denotation: quote symbol, string item sequence, quote symbol.

quote symbol: ".

string item sequence: string item; string item sequence, string item.

string item: ¢ any symbol except the quote symbol ¢; quote symbol, quote symbol.

unitary clause list: unitary clause; unitary clause list, semicolon symbol, unitary clause.

semicolon symbol: ;.

unitary clause: label, colon symbol, unitary clause; expression; jump.

colon symbol: :.

jump: goto symbol, label; label.

label: identifier.

goto symbol: GO, TO; GOTO.

expression: formula.

formula: formula, p1 operator, p2 operand; p2 operand.

TABLE XIX (Continued)

```
p2 operand:    p2 operand,    p2 operator,    p3 operand;    p3 operand.

p3 operand:    p3 operand,    p3 operator,    p4 operand;    p4 operand.

p4 operand:    p4 operand,    p4 operator,    p5 operand;    p5 operand.

p5 operand:    p5 operand,    p5 operator,    p6 operand;    p6 operand.

p6 operand:    p6 operand,    p6 operator,    p7 operand;    p7 operand.

p7 operand:    p7 operand,    p7 operator,    p8 operand;    p8 operand.

p8 operand:    p8 operand,    p8 operator,    p9 operand;    p9 operand.

p9 operand:    p9 operand,    p9 operator,    monadic operand;
     monadic operand.

monadic operand; monadic operator, monadic operand; primary.

primary: identifier; denotation; selector, primary; slice; particular
     program; standard procedure, left parenthesis symbol, parameter
     list, right parenthesis symbol; cast.

selector: RE; IM.

slice: identifier, sub symbol, indexer list, bus symbol.

sub symbol: (/.

bus symbol: /).

indexer list: indexer; indexer list, comma symbol, indexer.

parameter list: unitary clause list; paremeter list, comma symbol, uni-
     tary clause list.

begin symbol: BEGIN.

end symbol: END.

left parenthesis symbol: (.

right parenthesis symbol: ).

if symbol: IF.

fi symbol: FI.

then symbol: THEN.

else symbol: ELSE.
```

TABLE XX

PROGRAM BLOCKS

```
#  The following are closed clauses  #
BEGIN ... END
( ... )

#  The following are conditional clauses  #
IF ... THEN ... FI
IF ... THEN ... ELSE ... FI
CASE ... IN ... OUT ... ESAC
IF ... THEN ... ELSF ... THEN ... ELSE ... FI
( ... | ... | ... )
```

The subset includes only simple forms of declarations. The basic data types which are allowed are INT, REAL, COMPLEX, BOOL and CHAR. Also, identifiers may be declared as an array with any number of rows for the basic data types. Declarations may not contain expressions or initializations requiring mode coercion. Sample declarations are shown in Table XXI.

TABLE XXI

SAMPLE DECLARATIONS

```
INT I,J,K,M:=3;
REAL A,B;
COMPL Z;
(/ 0:10 /) INT ROW_OF_INTEGRAL;
(/ M,M /) CHAR ROW_OF_ROW_OF_CHARACTER;
```

Expressions comprise the remainder of the ALGOL 68 subset. The subset grammar describes the syntax of expressions but not the semantics. This is due to mode considerations which are not easily describable by the context free grammar. For example, the and operator (&) applied to

two complex values is syntactically correct, but is semantically undefined. A set of standard operators and procedures has been implemented for use in writing expressions. The standard operators are shown with their priorities in Table XXII. A list of the standard procedures which are implemented is given in Appendix A.

TABLE XXII

PRIORITIES FOR STANDARD OPERATORS

dyadic									monadic
1	2	3	4	5	6	7	8	9	(10)
-:=	OR	&	=	<	-	*	**	!	- + ¬ REPR
+:=			¬=	<=	+	/		?	ABS
*:=				>=		//			UPB LWB
/:=				>		//:			ODD SIGN ROUND
//:=									RE IM CONJ
//::=									
+=:									
:=									

APPENDIX D

A SAMPLE ALGOL 68 PROGRAM

```
#  THIS SAMPLE PROGRAM BUILDS A BINARY SEARCH TREE FOR INTEGRAL KEYS
   WHICH ARE READ FROM PUNCHED CARDS.  THREE ARE NO DUPLICATE KEYS AND A
   KEY OF ZERO INDICATES THE END OF THE INPUT DATA.

   DEFINITION OF VARIABLES--
        AVAIL-- THE AVAILABLE STORAGE POINTER
        KEY-- THE ARRAY OF KEYS IN THE TREE
        LLINK-- THE ARRAY OF LEFT LINK POINTERS
        NEW_KEY-- THE KEY BEING INSERTED
        RLINK-- THE ARRAY OF RIGHT LINK POINTERS
        ROOT-- THE ROOT POINTER                                          #

BEGIN
   (/ 200 /) INT LLINK, KEY, RLINK;
   INT ROOT:=0, AVAIL:=0;

   BEGIN
        INT NEW_KEY, I, J;

READ_AND_TEST:
        READ(NEW_KEY);

        #  TEST FOR ZERO END OF DATA INDICATOR  #
        IF NEW_KEY=0 THEN ¢GOTO¢ BUILT FI;

        #  TEST FOR FIRST KEY ENTRY  #
        IF ROOT=0 THEN KEY(/ ROOT:=AVAIL+:=1 /):=NEW_KEY;
                       LLINK(/ ROOT /):=RLINK(/ ROOT /):=0;
                       GO TO READ_AND_TEST
                       FI;

        I:=ROOT;

TRAVERSE:
        IF NEW_KEY>KEY(/ I /) THEN
             IF RLINK(/ I /)¬= 0 THEN
                       I:=RLINK(/ I /);
                       TRAVERSE
                  ELSE KEYS(/ RLINK(/ I /):=J:=AVAIL+:=1 /):=NEW_KEY;
                       LLINK(/ J /):=RLINK(/ J /):=0;
                       READ_AND_TEST
                  FI

             ELSF LLINK(/ I /)¬= 0 THEN
                       I:=LLINK(/ I /);
                       TRAVERSE
                  ELSE KEYS(/ LLINK(/ I /):=J:=AVAIL+:=1 /):=NEW_KEY;
                       LLINK(/ J/):=RLINK(/ J /):=0;
                       READ_AND_TEST
                  FI
```

```
BUILT:
     #   AT THIS POINT THE TREE HAS BEEN CREATED  #

     END

END
```

VITA

John Clute Jensen

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF A SCIENTIFIC SUBSET OF ALGOL 68

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Tulsa, Oklahoma, May 28, 1949, the son of Mr. and Mrs. Clute Jensen.

Education: Graduated from Tulsa Central High School, Tulsa, Oklahoma, in May, 1967; received Bachelor of Science degree in Mathematics from Oklahoma State University in 1971; completed requirements for the Master of Science degree at Oklahoma State University in July, 1973.