FASBOL, A SNOBOL4 COMPILER

by

P. J. Santos, Jr.

Memorandum No. ERL-M314

December 1971

(cover)

FASBOL, A SNOBOL4 COMPILER

by

Paul Joseph Santos, Jr.

Memorandum No. ERL-M314

December 1971

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ABSTRACT

The FASBOL compiler system represents a new approach to the processing and execution of programs written in the SNOBOL4 language. In contrast to the existing interpretive and semi-interpretive systems, the FASBOL compiler produces independent, relocatable assembly-language programs. These programs, using a small run-time library, execute much faster than under other SNOBOL4 systems.

With very little loss of flexibility, FASBOL offers the same advantages as other compiler systems, such as:

1. Up to two orders of magnitude decrease in execution times over interpretive processing for most problems.
2. Much smaller storage requirements at execution time than in-core systems, permitting either small partitions or larger programs.
3. Capability of independent compilation of different program segments, simplifying program structure and debugging and permitting overlays for very large programs.
4. Capability of interfacing with FORTRAN and machine-language programs, providing any division of labor required by the nature of a problem.

The FASBOL compiler is itself written in FASBOL, and implemented on the UNIVAC 1108 under EXEC II.

ACKNOWLEDGEMENT

The material presented here is the culmination of over two years of work, during which time I was assisted by several individuals and organizations.

Professor W. D. Maurer suggested the original research topic and provided guidance during the entire period. He served as an arbiter (and advocate) of practicality and suggested several ideas which were later incorporated into the final design. I received some financial support as his Research Assistant under NSF Grant GJ-821.

Information Systems Design, Oakland, provided extensive free computer time to develop and test FASBOL, and several members of the staff, mainly Messrs. D. Drake, G. Lutz, and K. Marcellius, helped with suggestions, criticism, and test applications.

Messrs. R. Karpinsky and R. Robertson helped in an unsuccessful but instructive attempt to bootstrap FASBOL using the 360 interpreter, and Mr. Karpinsky suggested the TIMER measurement feature.

Mrs. J. DeLaney typed this document, in very good time, from a barely legible hand draft.

Finally, my wife and family provided the all-around support without which I could never have accomplished this work.

TABLE OF CONTENTS

CHAPTER 1

Introduction

This chapter provides a brief description of the SNOBOL4 language, the relationship that FASBOL has to it, and an overview of the remaining chapters. In order to fully understand Chapters 2, 3 and 4, the reader must be thoroughly familiar with SNOBOL4; the brief description below is only intended to give an idea of the capabilities of the language and to refresh the memory of someone who has had some previous experience with it.

1.1 History and short description of SNOBOL4

SNOBOL4 is a high-level programming language incorporating non-numerical and list processing features not found in other languages. It permits the user to easily write programs containing string and symbol manipulation, pattern matching, and arbitrary data structure processing; it has applications in areas such as compiler writing, text processing, natural language recognition, and artificial intelligence. SNOBOL4 evolved from SNOBOL[1,2,3]*, developed at Bell Telephone Laboratories, Inc., in 1962, and is a widely known and used language [4]; it has been implemented on several different computers, including the IBM System/360, UNIVAC 1108, GE 635, CDC 3600, CDC 6000 series, PDP-10, SIGMA 5/6/7, ATLAS 2, and RCA SPECTRA 70 series.

---

*Numbers in brackets refer to references in the back.

One type of datum in SNOBOL4 is the character string, and the language provides operations for joining and separating strings, testing their contents, and replacing segments of them with other strings. For instance:

```
X = 'THIS COMPILATION WAS DONE ON '
```

assigns the string value (a literal of 29 characters) to the variable "X" . Given this assignment, a subsequent statement might be:

```
COMPILATION.MESSAGE = X DATE() ' AT ' TIME.OF.DAY()
```

where the expression "DATE()" is a function call which returns a string representing the date (similarly for "TIME.OF.DAY()"), and the operation performed is a concatenation of four strings to form a new one, which is assigned to the variable "COMPILATION.MESSAGE". This new string could be printed out and would appear as:

```
THIS COMPILATION WAS DONE ON Ø6 OCT 71 AT 19:5Ø:3Ø
```

Space need not be reserved ahead of time for dynamically varying quantities such as strings, and no type declarations need be given for variables, which may hold data values of any type (e.g., strings, integers, reals, etc.).

Another type of datum is the pattern, which is a specification of an arbitrarily complex structure for a character string; in order to test if a string fits a given pattern structure, a pattern match can be executed which either succeeds or fails (each statement provides for a separate transfer of control for either circumstance)

at the task. For instance:

```
FORTRAN.LABEL = (SPAN('0') | '') SPAN('0123456789') . LABEL
```

assigns to the variable "FORTRAN.LABEL" a pattern which both matches a string of digits, and assigns to the variable "LABEL" the matched string minus any leading zeros. This pattern might then be used in a pattern match statement:

```
CARD.IMAGE (FORTRAN.LABEL | '     ') TAB(5) =
```

which recognizes a properly formed or missing label field in a FORTRAN statement, truncates the first five columns off the string held in "CARD.IMAGE", and assigns the label, if any, to "LABEL".

SNOBOL4 provides for arithmetic on integers and real numbers, and for conversion between them and their string representation. Arrays may be created dynamically with any number of dimensions and any dimension bounds; each element of an array, like any variable, may hold a datum of any type, including another array. Functions may be dynamically defined and redefined, and function calls may be made recursively. An execution-time symbol table provides the capability of referencing a program entity (variable, label, or function) via the string representation of its name. Additional data types may be program-defined and used to form linked-list structures or extended elementary data, such as complex numbers.

Because SNOBOL4 has been implemented on many different computers, and new language features have been added as the language evolved, several different versions are currently in use. For the present purpose, the language is considered to be defined, both syntactically

and(almost completely) semantically, by the Prentice-Hall publication [5] which describes SNOBOL4, Version 2, as implemented on the IBM System/360. A later edition of the book (1971) describes Version 3; it appeared too late for inclusion in the present work, and in any case only contains some additional features which do not change the basic nature of the language.

1.2 Objectives of FASBOL

With one exception, all current implementations of SNOBOL4 are interpretive. A SNOBOL interpreter is best defined by its characteristics and mode of operation. The interpreter is a large program which processes the SNOBOL4 statements and translates them into some intermediate notation (e.g., prefix-polish with symbol table entries). After the entire program has been processed, the interpreter then enters the execution phase, where it acts as a table-driven processor to perform the operations specified by the SNOBOL4 program. Interpretive execution (as opposed to compiled execution) is slow and requires the interpreter program as well as the translated source program to be resident in core at execution time. Interpretive processing requires a translation for every execution (unless the process is "frozen" and saved), and must deal with a logically complete and self-contained SNOBOL4 program. One advantage over compilation is the greater flexibility provided at execution time; for example, a program can extend itself by processing strings into intermediate notation and then executing the newly created program.

The one exception (besides FASBOL) is a proprietary package for the IBM S/360 called SPITBOL[6] developed by one of the implementors of the SNOBOL4 interpreter. Although no information has been released about its internal workings, it appears to be a semi-interpretive system with a considerable execution speed advantage over regular interpreters for certain types of statements. It is also an "in-core" system and the speed advantage seems to be derived from the compilation into absolute machine code of the more conventional features (i.e., arithmetic) of SNOBOL4, and possibly from the transfer of control to the absolute program instead of the table-driven mode of the standard interpreter; however, it appears that patterns are generally still handled as they are by other implementations (i.e., as data structures to be interpreted). SPITBOL makes maximum use of the S/360 instruction set and its techniques are generally not machine-independent.

FASBOL* was conceived as a true compiler for SNOBOL4; this was achieved by discarding a very few, seldom used features (for example, the CODE datatype and function), and adding declarations. FASBOL has separate compilation and execution phases, just like FORTRAN; the compiler produces an independent, relocatable program; when the program is executed, it makes use of a library of run-time routines.

---

*The name "FASBOL", which has no other mnemonic significance besides the obvious one, is here used interchangeably to mean the dialect of SNOBOL4 processed by the FASBOL compiler, and the compilation system itself. A preliminary version, named SUBBOL [7,8], which did not implement all the current features but did do compilation of patterns, was started in the fall of 1969 and became operational June 1970.

The run-time library is small and modular; it provides predefined functions and entries to do operations not easily realizable by "in-line" code. One possible criticism of FASBOL as a compiler is that executing programs do spend a lot of the time in the library routines, and therefore it could be considered an interpreter with deferred execution. Although this criticism may be philosophically valid, in practice many "compiler" systems exhibit similar behavior (for instance formatted I/O in FORTRAN which could be compiled); FASBOL depends heavily on the library because most SNOBOL4 operations do not have counterparts in the instruction codes of digital computers, and "in-line" code would produce excessively large programs in return for a very small increase in running speed. All structural aspects of the source program, including patterns, are compiled into executable code; it is conceivable that on an imaginary computer whose instruction set included string manipulation and generalized stack operations, the library would be reduced to a few primitives and utility routines such as free-storage and I/O.

The primary design objective for FASBOL was to produce fast-executing programs. The speed advantage over interpreters is due to several factors. Because control resides in the compiled program, the "middleman" that does the interpretation is eliminated. This effect is accentuated in the case of pattern matches, which FASBOL executes as a series of re-entrant subroutines, but are interpretations within-an-interpretation on other systems. Because the types of some variables can be declared, some arithmetic can be executed, in-line, FORTRAN style, rather than as a series of

type-checks required by the typeless nature of variables in SNOBOL4. Because there is no resident processor (the entire FASBOL library, including all primitives, is less than 5K), the free storage package has more space available for dynamic storage allocation, which improves the speed of execution. SPITBOL shares some of the above advantages with FASBOL, but cannot produce FORTRAN-like arithmetic, does not compile patterns, and uses more storage (but not as much as the interpreters) for itself at run-time.

The execution speed of FASBOL has been compared with that of four other SNOBOL4 systems, using the absolute times to execute statements in three categories: GENERAL, ARITHMETIC, and PATTERN (the first covering operations other than the latter two). These four systems are the UNIVAC 1108 interpreter, the CDC 6400 interpreter, the interpreter for the IBM System/360 (Model 65), and SPITBOL (also on the 360/65).

The UNIVAC 1108 interpreter, implemented at the University of Maryland from the macros supplied by the Bell Telephone Laboratories (used also for the 360 interpreter), runs under the EXEC II operating system with a 65K of machine memory. It is a full implementation of SNOBOL4, Version 2, and provides a direct basis for comparison with FASBOL since it runs on the same machine under the same conditions. The various categories of statements were timed by measuring the elapsed time for a number of iterations of the statement, and then dividing by the number of iterations. The results indicate that FASBOL is roughly 200 times faster on arithmetic, 20 times faster on patterns, and at least 10 times faster on other types of operations.

The interpreter for the CDC 6400 was implemented at the University of California at Berkeley and is a carefully hand-coded version which is not a full implementation of SNOBOL4 but does an extremely efficient job of processing large numbers of student programs at the University Computer Center. Since actual elapsed time was not necessarily spent in SNOBOL4 execution, statement times were measured by running two jobs that were incrementally different, and then using time difference in chargeable CPU time (divided by the number of additional iterations). This method also factored our the "compilation" time, which is not reported separately. In this comparison, FASBOL was approximately 100, 10, and 5 times faster on arithmetic, patterns, and other operations, respectively. An additional problem here (and with the S/360) is to factor out the effects due to differences in the computers themselves. Both the 1108 and 6400 are word-addressable with roughly equivalent instruction execution times, but the 6400 has larger words (60 bits vs 36 bits), whereas the 1108 has a capability of referencing a partial word in memory (which helps in dealing with packed character strings). Since each machine has its own advantages, it is unlikely that hardware differences could affect the SNOBOL4 execution times by more than a factor of 2 in either direction.

The IBM System/360 has a distinct advantage over the other two computers when processing strings of characters; in addition to being byte-addressable, it has specific instructions to manipulate and compare character strings. Whereas the Model 65 has

instruction execution times comparable to the 1108, certain operations frequently used in string manipulation and pattern matching can be done considerably faster. As a typical case, consider the operation of moving a portion of a string from one position in memory to another, as would occur during a string concatenation. Assume that there are N characters to be moved from the source string pointed to by R1 starting at character position P1, to the object string pointed to by R2 starting at position P2. This can be accomplished by the single 360 instruction

MVC P2(N,R2) , P1(R1)

Where the total time in microseconds is given by the formula: 2.93 + .38N; N is the number of bytes moved. The 1108 must convert the character positions to full words plus remainders, and use an explicit loop involving execute tables and two register pointers for each string. Assuming N,P1, and P2 are in registers A6,A0, and A2, respectively, the following sequence is required:

Instruction		Comment	Timing (μsec)
DSA	A0,36	. Form Source	.875
DI,L	A0,6	. Pointer	10.125
A	A0,R1	.	.75
DSA	A2,36	. Form Object	.875
DI,L	A2,6	. Pointer	10.125
A	A2,R2	.	.75
LXI,L	A0,1	. Set Word Pointer	1.00
LXI,L	A2,1	. Increments	1.00
LXI,L	A1,1	. Set Character Pointer	1.00
LXI,L	A3,1	. Increments	1.00

	Instruction		Comment	Timing (μsec)
	AN,L	A6,1	. TOT Chars - 1	.75
LOOP	EX	GTCHR1,*A1	. Get Source Char,Bump PTR	(.75+1.17)N
	EX	PTCHR2,*A3	. Put Object Char,Bump PTR	(.75+1.17)N
	JGD	A6,LOOP	. LOOP for each Char	1.5(N-1)+.75

The execute tables, indexed by the character pointer, are as follows:

GTCHR1	L,C5	A5,0,A0	. Get First Char	.75	(with prob.1/6)
	L,C4	A5,0,A0	. " Second "	.75	"
	L,C3	A5,0,A0	. " Third "	.75	"
	L,C2	A5,0,A0	. " Fourth "	.75	"
	L,C1	A5,0,A0	. " Fifth "	.75	"
	LMJ	B11,$+1	. Save Execute Loc	.875)	
	L,C0	A5,0,*A0	. Get Sixth Char	.75)	"
	LXM,L	A1,0	. Bump Word PTR,Reset	.875)	
	J	0,B11	. Char PTR, Return	.75)	

Since the probability of any character position is 1/6, the average time for one execute table reference is 7.00/6 = 1.17 μsec (as used in the sequence above). PTCHR2 is similar except that it stores characters instead and uses the register set (A2,A3) for word and character pointers. The total time in microseconds is given by the formula: 27.5 + 5.34N ; thus even the incremental speed factor of the 360 over the 1108 in this case is 5.34/.38 = 14. Similar advantages occur in string comparisons, which are quite frequent during pattern matching.

The execution times for statements under the interpreter and SPITBOL on the 360/65 were measured in a similar fashion as on the 1108, and indicated that FASBOL is faster than the interpreter (which is the definitive full implementation of SNOBOL4) by the

same ratios as with the 6400 interpreter (100, 10, and 5). This is only twice as fast as the 1108 interpreter, and is indicative of the fact that these interpreters are both implemented from macros in order to provide machine-independence so that the 360 version may not make full use of the instruction set available to it. SPITBOL, on the other hand, does make full use of these instructions and also avoids interpretive mode execution to some extent. FASBOL is 4-5 times faster than SPITBOL on arithmetic, but roughly equal in most other operations. Complicated patterns run faster in FASBOL because SPITBOL cannot optimize them and reverts back to an interpretive mode. Very simple patterns and string concatenations run faster in SPITBOL, where the 360 instructions outperform their 1108 equivalents, as illustrated by the above example of a string move. The FASBOL compilation techniques could be applied on the 360 as well, in addition to the optimizations performed by SPITBOL, resulting in even faster execution.

In addition to meeting the primary objective FASBOL has other characteristics, common to compiler systems (and absent in the SNOBOL4 interpreter and SPITBOL), which may be as important as fast execution for some problems. FASBOL is capable of compiling separately main programs and independent subroutines, with the option of making variables, labels, and functions either local or global. This facility allows for the fragmentation of an otherwise insoluble problem into manageable pieces which can be checked out before being joined. It also permits an overlay structure in storage for problems that would otherwise not fit or leave very little room for free storage. FASBOL programs may call on, and more important, be called

by FORTRAN or assembly-language routines; a mixed system of FORTRAN and FASBOL routines is ideal for dealing with a problem involving both numerical and non-numerical processing. Once a FASBOL program is debugged, it need never be recompiled.

The interpreters (and SPITBOL) do have advantages over FASBOL in some cases. Dynamic compilation is of course not allowed in FASBOL; neither are other highly permissive features such as the redefinition of operators. FASBOL has no explicit tracing capability, sometimes making debugging more difficult. FASBOL has no particular advantage in certain environments where every execution is preceded by a compilation, as frequently happens with runs submitted by students for a programming class, and for short-run programs.

1.3 Brief Outline of Remaining Chapters

Although currently implemented on a UNIVAC 1108, the run-time techniques and compilation schemes used by FASBOL are largely machine-independent. Certain features of the SNOBOL4 language which provided more flexibility than the compiler could efficiently deal with are not implemented; they are generally not used by most SNOBOL4 programs. FASBOL provides both compile-time and run-time features absent from other implementations, including additional predefined functions for programming convenience and efficiency of operation. Chapter 2 provides a full description of all deletions, changes, and additions to SNOBOL4 in FASBOL, and (in conjunction with the SNOBOL4 manual)serves as a user's guide to FASBOL programming. Chapter 3 describes the important parts of the run-time system in reasonably machine-independent fashion, and Chapter 4 gives a general functional

description of the compiler, which is itself written almost exclusively in FASBOL.

The chief unimplemented feature of SNOBOL4 is the run-time compilation capability, involving the EVAL and CODE primitives, direct GOTOS, and the CODE and EXPRESSION datatypes. The only other major unimplemented feature is the tracing capability. Features that are implemented differently are small in number and of minor importance (for instance, the character set differs in three characters, and &MAXLNGTH is initially set to 32767). In addition to some minor extensions to the SNOBOL4 syntax, extra features of FASBOL include declarations and compiler options, more control cards, and more primitives (pre-defined functions). The declarations can be used to optimize the operation of the compiled program and/or permit communication with other programs; the options provide user control over certain run-time parameters or compiler operation. The additional control cards permit more listing control, cross-referencing capability, etc., and the additional primitives include routines for extracting or inserting substrings, performing logical operations on integers, and controlling run-time parameters.

The FASBOL run-time operation is controlled by the compiled programs which make use of the system library for repetitive and/or global functions. Expressions fall into three categories in terms of the code generated for them. Arithmetic expressions involving only constants and dedicated variables (see declarations) or functions are compiled directly into the equivalent machine-language instructions. Other expressions which are not explicitly pattern

structures are compiled as a postfix polish series of evaluations using a single expression stack for temporary results. Explicit pattern expressions are compiled as two separately executable phases: an evaluation phase to evaluate any elements of the pattern, using either of the above code conventions, and a match phase, coded as a re-entrant subroutine reflecting the structure of the pattern and using two stacks operating in orthogonal sequence.

The FASBOL compiler is a one-pass processor which generates a symbolic assembly language program and then automatically invokes the assembler. It is structured in space/time as a main program with five phases which successively overlay each other in main storage. The main program provides the control to pass from one phase to the next, plus any subroutines needed by more than one phase. The phases perform in succession, source input editing, declaration processing, executable statement processing, program storage and data allocation, and (optionally) cross-reference dictionary listing.

CHAPTER 2

FASBOL Programming

This chapter provides all the additional information required for writing FASBOL programs, given familiarity with SNOBOL4, but does not cover the SNOBOL4 language itself. Most SNOBOL4 (Version 2) programs will run "as-is" under FASBOL, or will require only the addition of some of the declarations described in 2.2.1. It is anticipated that there will be future versions of FASBOL which will incorporate additional features, such as some of the features of SNOBOL4, Version 3.

2.1 General Language Features

FASBOL programs are compiled into symbolic assembly language programs and assembled to produce relocatable object programs. The intermediate symbolic program may be saved, optimized, and then assembled by the experienced user who wishes to do so, and can answer for his handiwork. The source statement for a given section of assembly code is inserted as a comment line immediately preceding that section. A future feature of FASBOL compilation will be direct generation of relocatable object programs for improved efficiency.

FASBOL programs are executed by calling on the allocator (linking loader) to produce an absolute element from the relocatable FASBOL main program, subroutines, and library, and any other non-FASBOL routines used. The absolute (non-relocatable) element can then be immediately executed or saved for future use.

2.1.1 SNOBOL4 Features Not Implemented

A detailed list of unimplemented features is given in Table 1. The reason for most of these is the inability of the compiler to control, or be present, during execution; this is the case with items 1, 2, and 4. Other items, such as 3, 5, 6, 7, and 10, are unimplemented because they would require a much larger run-time symbol table and/or induce far less efficient code; also, they are seldom used. The remaining items are due to the nature of the run-time code generated for patterns.

2.1.2 SNOBOL4 Features Implemented Differently

Table 2 is a list of features implemented differently in FASBOL. Except for item 1, they have no major semantic effect.

Item 1 rectifies a quirk in SNOBOL4 which returned a STRING datatype as a result of the unary . (name) operation on a natural variable, instead of a NAME datatype. The FASBOL convention is more consistent and useful; it is noted that the authors of SPITBOL[6] decided on the same convention. NAME datatypes are useful in retaining access to a natural or created variable that would otherwise require a new indirection (symbol-table look-up)or array/field reference. To access or store into a variable pointed to by a NAME datatype, the $ (indirection) operator must be used. For instance, if

```
N = .ARRAY<25>
```

,then, to increment the value of the array element,

```
$N = $N + 1
```

TABLE 1

Unimplemented Features of SNOBOL4 *

1. EVAL(), CODE() , and direct Gotos.
2. Datatypes CODE and EXPRESSION.
3. Non-literal prototypes for DEFINE() and DATA().
4. All features, functions, and keywords having to do with tracing.
5. Predefined VALUE() field.
6. Redefinition, OPSYN(), or APPLY() of primitives and fields.
7. Indirection or I/O association for keywords.
8. Two features of QUICKSCAN mode:
   A. Continual comparison of the number of characters remaining in the subject string against the number of characters required.
   B. Assumption that unevaluated expressions must match at least one character.
9. Explicit pattern structures in parameter lists (except as an argument to ARBNO)or as an argument to unary *
   A. An explicit pattern structure is an expression containing at least one of the following:
      (1) Binary operator !, $, .
      (2) Unary operator *, @
      (3) Primitive pattern variable
      (4) Primitive pattern function
   B. The same effect can be achieved by assigning the structure to a variable and then using the variable in the parameter list, or as the argument to *.

*Version 2

10. &ARB, &ABORT, &BAL, &FAIL, &FENCE, &REM, &SUCCEED, and &DUMP keywords.

TABLE 2

Differences From SNOBOL4 *

1. Unary . always returns a value of type NAME, never a STRING for natural variables. Indirection(unary $) applied to a value of the type NAME returns the same as value. NRETURN is unnecessary and equivalent to RETURN. A NAME cannot be generated for a label or a function.

2. The third argument to OUTPUT() is an association length (as in INPUT()), rather than a FORMAT string.

3. The 1108 character set requires three characters to be different:

360 characters	029 code	1108 character	029 code
"	7-8	#	3-8
¬	11-7-8	\	11-7-8
|	12-7-8	!	11-0

4. Compiler-generated statement numbers are always listed on the left.

5. &MAXLNGTH is initially set to 32767.

6. If &ABEND is nonzero at program termination, it causes a DUMP of free storage only; any post-mortem dump (PMD) is the responsibility of the user.

7. APPLY() will accept either more or fewer arguments than the user-defined function being called, and rejects/null fills formal arguments accordingly.

---

*Version 2

; the proper context for NAME Datatypes is specified by the syntax in Appendix 1.

2.1.3 Additions to SNOBOL4

The chief additions, besides minor extensions to the syntax, are the implementation of declarations and compiler options, a full set of control cards, and more pre-defined (primitive) functions, all of which are described in the following sections. If declarations and/or options are used, they must precede all executable statements in the program.

There are three additions to the syntax of SNOBOL4:

(a) Quoted strings may be continued onto a new line, but note that the continuation character (. or +) is replaced by a single blank.

(b) Comment and control lines (* and -) may start inside a line image (after ;), and consume the remainder of the line image.

(c) The syntax of the DEFINE and DATA prototypes has been loosened to conform with the rest of SNOBOL4 syntax, with blanks allowed after (, around , , and before) .

The full syntax for FASBOL, including the declarations and options, is given in Appendix 1.

2.2 Declarations

The set of declarations in FASBOL have been divided into "Required" and "Optional" classes, the difference being that some SNOBOL4 programs must use some of the "Required" declarations in order to function properly. The general form of a declaration is

a call on the pseudofunction DECLARE, with two to four arguments. The first argument is always a string literal specifying the type of declaration, and the other arguments specify the parameters upon which the declaration has effect. The following descriptions are by declaration type, and contain examples of all possible forms of the declaration.

2.2.1 Required

2.2.1.1 INDIRECT Declarations

There are six forms of the declaration: INDIRECT.VARIABLE, INDIRECT.LABEL, INDIRECT.FUNCTION, NOT.INDIRECT.VARIABLE, NOT.INDIRECT LABEL, and NOT.INDIRECT FUNCTION. One deliberate incompatibility between the interpreter and FASBOL is that, whereas the interpreter retains a full symbol table for the program during execution, FASBOL provides only a run-time symbol table (if the program requires one); this method is fully compatible with SNOBOL4 programs provided the only entities accessed via indirection (unary $) are created variables. However, any natural variable, label, or function that at some time during execution will be referenced via a string (identical to its name),must generally (exceptions noted below) be declared INDIRECT, so that it can be explicitly inserted in the run-time symbol table by the compiled code. All possible cases where these declarations must be made are given here.

(a) Natural variables referenced via the $ operator. For example, if

```
A = 25 ; X = 'A'
```

then

```
EQ($X,25)
```

will succeed only if "A" has been declared INDIRECT, as in

```
        DECLARE('INDIRECT.VARIABLE','A,B,C,D')
```

On the other hand, none of the following uses of $ require declarations, provided they are intended to refer only to created variables:

```
        $INPUT  =  'RANDOM.CREATED.VARIABLE'

PUSH.STACK      I  =  I  +  1

           $('STACK' I)  =  VAL

POP.STACK       VAL  =  $('STACK' I)

              I  =  I  -  1

         $'F(X,Y)'  =  F(X,Y)
```

(b) Labels referenced via the $ operator, except if the operand is a string literal. The latter case is because it usually represents a simple GOTO to a label that does not satisfy the syntax rules for identifiers. For example, if

```
        I  =  J  -  (J / 4) * 4   : ($('CASE.' I))
```

```
CASE.0
CASE.1
CASE.2
CASE.3
```

Then there must be a declaration

```
        DECLARE('INDIRECT.LABEL','CASE.0 CASE.1 CASE.2 CASE.3;')
```

; on the other hand,

```
                                          :($'>>>1')

                          .
                          .
                          .

>>>1
```

Requires no declaration.

(c) Labels referenced via non-literal second arguments to DEFINE, as in

```
DEFINE('F(X,Y,Z)', 'CASE' I)
```

(d) Natural functions referenced via non-literal first arguments to APPLY, or non-literal first or second arguments to OPSYN. For example, if

```
DEFINE('F(X)')
Y = 'F'
```

Then

```
APPLY(Y,A)
```

will work (i.e., be equivalent to F(A) only if "F" is declared INDIRECT, as in

```
DECLARE('INDIRECT.FUNCTION', 'F,G,H')
```

; on the other hand,

```
OPSYN('G','F')
```

requires no declaration.

(e) Natural variables referenced via non-literal first arguments to INPUT, OUTPUT, and DETACH. For instance,

```
X = 'INPUT' ; INPUT(X,5,72)
```

requires "INPUT" to be declared INDIRECT, whereas

```
DETACH('INPUT')
```

does not.

If it is more convenient to declare all variables (or labels, or functions) INDIRECT, the second argument to the declaration is the dummy variable "ALL" :

```
DECLARE('INDIRECT.LABEL', ALL)
```

; exceptions can then be made using the NOT.INDIRECT declarations, such as

```
DECLARE('NOT.INDIRECT.LABEL','LAB1  LAB2  LAB3;')
```

It should be noted that the quoted parameter list for label declarations uses blanks for separators and a semicolon preceding the closing quote, the latter also used in other declarations where labels appear.

2.2.1.2 Other Required Declarations

These are the INPUT.ASSOCIATIONS, OUTPUT.ASSOCIATIONS, and FIELD declarations, and may be required for reasons given above and/or because of the one-pass nature of the compiler. The cases in which these three declarations must be made are, respectively:

(a) Natural variables which either occur as non-literal first arguments to INPUT or are referenced in the compilation sequence before their appearance in an INPUT call. For example, if

```
X = 'INPUT1' ; INPUT(X,5,73)
```

and

```
READ.IN = INPUT2
          .
          .
          .
INPUT('INPUT2', 5, 60')
```

then the variables must be declared by

```
DECLARE('INPUT.ASSOCIATIONS','INPUT1,INPUT2')
```

In the above example, "INPUT2" need only be declared if the execution sequence will pass through the INPUT call before using

"INPUT2" as an input-associated variable in a statement preceding the INPUT call. The compiler must know, when it encounters a variable, if there is a possible input association for it, and this information can only come from instances of literal first arguments to INPUT, or from the declaration.

(b) Natural variables which occur as non-literal first arguments to OUTPUT. Similar to (a), but not constrained by the one-pass problem.

(c) Field names (of programmer-defined datatypes) which are referenced in the compilation sequence before they appear in the argument to some DATA call. For example

```
NEXT(A)  =  TOP(B)
          .
          .
          .
DATA('BLOCK(NEXT,TOP,MIDDLE,BOTTOM)')
```

requires

```
DECLARE('FIELD','NEXT,TOP')
```

Since the usual programming practice is to place the I/O association and datatype definition statements at the beginning of a program (they must be executed before their respective effects are usable), these declarations are seldom actually needed.

2.2.2 Optional

The remaining declarations provide more flexibility for the programmer in specifying inter-program communication or direct the compiler to perform optimization of some of the generated code.

2.2.2.1 Dedicated Declarations

Variables can be dedicated to certain data values by declaring them as STRING, INTEGER, REAL, or PATTERN. Normally, values in FASBOL are represented by descriptors for one of the seven basic types of data (STRING, INTEGER, REAL, NAME, ARRAY, PATTERN, and DATA), and variables correspond to single storage locations containing such descriptors. Whenever a value is used in an operation, it generally must be checked for correct type (and possibly converted) before being used. Frequently, however, certain variables have a "dedicated" use, i.e., they are always assigned only one type of value, such as INTEGER; this is always the case in FORTRAN. By passing this information to the compiler, expressions in which these variables appear can be optimized.

INTEGER and REAL dedicated variables correspond to single storage locations which are used to hold an actual integer or floating-point number. The compiler generates FORTRAN-like code for expressions involving these variables; either as a subexpression for a larger descriptor-mode expression, or as an entire statement. In addition to simple arithmetic involving dedicated variables and constants, FORTRAN function calls also return dedicated values, as do the predicates LT, LE, EQ, NE, GE, GT and the primitives NOT, AND, OR, XOR, LSHIFT, and RSHIFT with dedicated integer arguments. For example, the statement

```
I = LE(I,100)  I + AND(RANDOM,1)
```

will be executed as a completely dedicated statement and about ten to one hundred times faster by the additional declaration

```
DECLARE('INTEGER', 'I,RANDOM')
```

PATTERN dedicated variables are similar to undedicated variables, except that they may only be assigned pattern descriptors or the null string descriptor. This dedication is natural for a variable that is used as a shorthand notation for some pattern structure, and serves to optimize the code generated when that variable appears as part of another pattern, as in

```
ABC  =  A | B | C
STRING  Z ABC  X  | X  ABC  Y  | Y  ABC  Z
```

with declaration

```
DECLARE('PATTERN', 'ABC')
```

STRING dedicated vatiables are useful only in passing string arguments to or from FORTRAN programs, and are less efficient than undedicated variables otherwise. A string variable is declared with a maximum character count, and corresponds to a series of storage locations containing, respectively, the maximum character count, the current character count, and the string itself. The FORTRAN interface provides a pointer to the first word of the string, which can be considered an array in "A6" format. An example of a STRING declaration is

```
DECLARE('STRING', 'CARDIMAGE(80),NAME(6),LETTER(1)')
```

Type-checking and/or conversion is automatically performed when values of descriptor mode must be loaded from or stored into dedicated variables, or when a dedicated expression appears inside a descriptor-mode expression. Dedicated variables cannot have I/O associations.

2.2.2 SNOBOL.SUBPROGRAM Declaration

This declaration indicates that the program being compiled is an independent subprogram, not the main program. The second argument is a quoted identifier conforming to the syntax for external symbols which is used to name the subprogram; it is an entry point with no arguments which performs any initialization required by INDIRECT declarations or the TIMER option (see 2.3) for the whole subprogram, then returns. If a subprogram uses neither of the above, any other entry point in it may be called without ever calling the subprogram entry; otherwise, the subroutines will execute without benefit of the INDIRECT and/or TIMER declarations until the subprogram entry is called. For example, if a subprogram contained a set of entries (see 2.2.2.4) to do complex arithmetic (i.e.,CMPADD, CMPSUB, etc), it might be declared

```
DECLARE('SNOBOL.SUBPROGRAM','COMPLX')
```

where the entry COMPLX need never be called, provided there are no INDIRECT/TIMER declarations. On the other hand, if the subprogram does use either of these, the call

```
COMPLX()
```

should be executed first, by some program before any other entries in the subprogram are called.

2.2.2.3 EXTERNAL Declarations

Items that exist in some external program can be declared using EXTERNAL.VARIABLE, EXTERNAL.LABEL, and EXTERNAL.FUNCTION declarations; their names must conform to the syntax for external symbols. The purpose of these declarations (and those of 2.2.2.4) is to allow

communication between separately compiled FASBOL programs by defining certain variables, labels, and functions on a global level, rather than on a local level for each subprogram. For example

```
DECLARE('EXTERNAL.VARIABLE', 'ALPHA,ALPHA1')
DECLARE('EXTERNAL.LABEL', 'XTL1  XTL2  XTL3;')
DECLARE('EXTERNAL.FUNCTION', 'CMPADD,CMPSUB,CMPMUL, CMPDIV')
```

implies that all the items declared exist in some other program, and they are referenced by their external names when they appear in the code. (Rather than by their compiler-generated internal name if they were local to the subprogram.)

2.2.2.4 ENTRY Declarations

These are the logical counterparts to the EXTERNAL declarations. For example,

```
DECLARE('ENTRY.VARIABLE', 'ALPHA,ALPHA1')
DECLARE('ENTRY.LABEL', 'XTL1  XTL2  XTL3;')
DECLARE('ENTRY.FUNCTION', 'CMPADD(ARG1,ARG2)')
DECLARE('ENTRY.FUNCTION','CMPSUB(ARG1,ARG2)')
DECLARE('ENTRY.FUNCTION','CMPMUL(ARG1,ARG2)')
DECLARE('ENTRY.FUNCTION','CMPDIV(ARG1,ARG2)')
```

complement the declarations of 2.2.2.3. The variables and labels declared here are local to this subprogram, but can also be referenced globally via their external names. The same is true of the functions, but in addition the declaration has the effect of predefining the function, so that no DEFINE call is needed for that function. It should be noted that only one function can be defined by each declaration, and that there is an optional third argument

(like the optional second argument to DEFINE) which specifies a starting label, as in

```
DECLARE('ENTRY.FUNCTION','F()LOC1,LOC2', 'START;')
```

2.2.2.5 FORTRAN Interface Declarations

FORTRAN functions (or subroutines) that are to be called by a FASBOL program are specified using the EXTERNAL.FORTRAN.FUNCTION declaration, which in addition gives the type (integer or real) of each function either implicitly (by FORTRAN conventions, if the first letter of the name is I through N, it is an integer, and real otherwise), or explicitly. For example,

```
DECLARE('EXTERNAL.FORTRAN,FUNCTION','IFUNC,RFUNC,QFUNC=
INTEGER,KFUNC=REAL')
```

declares IFUNC and QFUNC to be integer-valued FORTRAN functions and RFUNC and KFUNC to be real-valued FORTRAN functions. In addition, if there is any possibility that a FORTRAN program, after being called from FASBOL, will in turn call a FASBOL program, it must be declared 'OPEN'. For example,

```
DECLARE('EXTERNAL.FORTRAN.FUNCTION','F,G','OPEN')
```

implies that within the FORTRAN functions F and G there are calls on FASBOL programs.

An entry into FASBOL that is to be called from FORTRAN is specified using the ENTRY.FORTRAN.FUNCTION declaration. This declaration also serves to declare the type(implicitly or explicitly) of the function value, and the types and number of formal arguments to the function. In addition, a third argument specifies a unique label,

associated with the declaration, to which control is transferred in order to return to the FORTRAN program. An optional fourth argument specifies a starting label,if different from the function name. For example, the declaration

```
DECLARE('ENTRY.FORTRAN.FUNCTION','F=INTEGER(I,J=REAL,K(6))',
.'FRET;','FSTART;')
```

specifies an integer-valued function F callable from FORTRAN, which starts at label FSTART, and which returns when a transfer to FRET occurs. It has three formal arguments: an integer, a real, and a string with a maximum length of six characters. A side-effect is that F and I are dedicated integer variables, J is a dedicated real variable, and K is a dedicated string variable. The maximum character count for K may be set higher by a separate STRING declaration. The function variable (in the above example, F) can only be of type INTEGER or REAL.

2.3 Options

Options are conditions which affect the operation of the running program in a global or subroutine-by-subroutine way. Except for NO.STNO and TIMER, they may only be stated in the main program. The general form of an option is a call on the pseudofunction OPTION with one of the following arguments:

'QUICKSTORE' speeds up the free storage mechanism by making each acquired block non-returnable. This is usable only if the program does not acquire more than the available storage before terminating.

'NO.LOOK-AHEAD' disables double-buffering on drum and tape input; necessary only if some non-SNOBOL I/O operation is concurrent on the

I/O unit.

'HASHSIZE=N' overrides the standard bucket table size (64, for N=6). N must be between 1 and 13 and is the bit width of the hashed index for the symbol table. Useful for reducing bucket table size in programs that barely do (but do) use it, or for enlarging it in programs that make extensive use of it.

'NO.STNO' sets a mode in the compiler that cuts out the bookkeeping on &STNO, &LASTNO, and &STCOUNT; it makes an appreciable difference in dedicated arithmetic statments.

'TIMER' sets a mode in the compiler that causes timing statistics to be kept for each statement and printed out when the execution terminates. This mode can be set selectively in the main program and subroutines. The time expended by a function call is included in the statement in which it occurs. TIMER and NO.STNO are imcompatible. Intermediate timing statistics can be produced for any program by calling the primitive EXTIME.

2.4 Control (in addition to -LIST and -UNLIST from SNOBOL4)

-NOCODE, -CODE turn off and on, respectively, the listing of object code between statements. The normal mode is off (NOCODE).

-EJECT causes a page eject in the source listing, and inserts an eject control in the object code.

-FAIL, -NOFAIL* turn off and on, respectively, a feature which

*Full credit for this idea goes to the authors of SPITBOL who also inspired the inclusion of the primitives DUPL,LPAD,RPAD, and REVERS (next section).

traps unexpected statement failures. When the feature is on (NOFAIL), any statement within its scope that does not have a conditional GOTO and which fails, will cause an error exit. Note that an unconditional GOTO is equivalent to none at all, and will be trapped if the statement fails. Normal mode is off (FAIL).

-NOCROSS,-CROSSREF turn off and on, respectively, the cross-referencing capability of the compiler. The normal mode is off (NOCROSS), and no cross-reference listing is produced unless it is turned on at least once. Either the entire program or selected parts can be cross-referenced.

-NOASM, -LISTASM control the post-compilation assembly, which is normally done with listing turned off. NOASM prevents assembly, and LISTASM causes it to be done with listing turned on.

-SPACE [N] spaces N (or 1, if missing) lines in the source listing.

-NEWSTNO N resets the statement number to any value (N) greater than 0.

2.5 Primitives (* indicates new primitives)

2.5.1 Pattern Primitives

(a) *BREAKX(String)

Equivalent to: BREAK(String) ARBNO(LEN(1) BREAK(String));
but faster and less space-consuming.

(b) *NSPAN(String)

Like SPAN but succeeds even if no characters are spanned.

(c) *STRVAL(String)

Used to (1) induce string concatenation within a pattern

expression, and/or (2) optimizes a pattern expression by specifying a given element to have a string value.

2.5.2 Expression Primitives

(a) CONVERT(any, string)

Now will also convert reals to integer by truncation.

(b) *EJECT()

Causes page eject on printer, returns null.

(c) COLLECT(integer)

Sets the blocksize (Default=128) at which the free storage mechanism releases accumulated unused blocks and merges them back together if possible. The trigger is set whenever a block is requested that causes free storage to use or split a block equal to or greater than the collect size. The qualitative effect of setting the collect size to a small value is to minimize free storage fragmentation but causes frequent collections, whereas the effect of setting it to a large number minimizes the collection overhead but leads to more storage fragmentation (which may be desirable), and possibly premature storage overflow (which is not). Returns null.

(d) *BUFSIZ(integer)

Sets the maximum number of words+1(Default=136) allowed to be used for output buffering of tape and drum records. For instance, the Default allows for 9 ((14+1)*9=135) 14-word records (card image size) and 5 ((22+1*5=115) 22-word records (print image size) on each active tape or drum unit. Returns null.

(e) *SUBSTR(string,integer,integer)

Returns a substring of the first argument starting at the position specified by the third argument (same conventions as POS primitive), of length specified by the second arg, and fails if it is not a proper substring. For example: SUBSTR('ABCDEFG',3,1) ⇒ 'BCD', and SUBSTR('ABCDEFG',4,4) fails. This is equivalent to, but much faster and less space consuming than the pattern TAB(INTEGER) LEN(INTEGER).VAL

(f) *INSERT(string,string,integer,integer)

Returns the string formed by replacing the substring specified by the last three args by the first in the second arg, or fails if not a proper substring. Example: INSERT('A','SYNONYMOUS',2,0) ⇒ 'ANONYMOUS';equivalent pattern: STRING2 TAB(INTEGER) . PART1 LEN(INTEGER) REM . PART2 ; VAL = PART1 STRING1 PART2

(g) *DUPL(string,integer)

Returns the string formed by duplicating the first arg the number of times specified by the second arg

(h) *LPAD(string,integer,string)

Returns the string formed by padding the first arg on the left with the character specified by the third arg to a length specified by the second arg. If the first arg is already too long, it is returned unchanged. If the third arg has more than one character, the rest

are ignored. If the third arg is null, blanks are used.

(i) *RPAD(string,integer,string)

Same as LPAD but pads to the right.

(j) *REVERS(string)

Returns the string formed by reversing the order of the characters of its argument, e.g., REVERSE('ABC') ⇒ 'CBA'.

(k) *AND(integer,integer)

Returns the integer formed by the 36-bit logical AND of the two arguments.

(ℓ) *OR(integer,integer)

36-bit logical OR.

(m) *XOR(integer,integer)

36-bit exclusive OR.

(n) *NOT(integer)

36-bit one's complement

(o) *RSHIFT(integer,integer)

36-bit logical right shift of first arg by second.

(p) *LSHIFT(Integer,integer)

36-bit logical left shift of first arg by second.

(q) *EXTIME(string)

Prints out timing statistics for program whose name is given by the argument, and returns null, or fails if program is not being timed.

(r) *REMCOR()

Returns an integer which is the total amount of free storage remaining unused.

2.6 FASBOL Programming Techniques

Most of the programming techniques described in the SNOBOL4 manual [5] apply to FASBOL programs as well, with a few exceptions described below. In addition, several new techniques should be noted when writing FASBOL programs, due to the greater opportunity to optimize code.

2.6.1 Dedicated Expressions

Arithmetic in FASBOL is most efficient when as much of it as possible is performed in dedicated (non-descriptor) mode. The rules for forming dedicated real expressions are:

(a) A dedicated real variable, a real constant, or a real FORTRAN function call is a real expression.

(b) An arithmetic expression consisting of real expressions, unary + and -, binary +, -, *, and /, and parentheses is a real expression.

(c) An arithmetic expression consisting of at least one real expression, undedicated variables with no input associations, the above operators, and parentheses is a real expression.

The rules for forming dedicated integer expressions are analogous to the above, plus:

(d) An arithmetic expression consisting of integer expressions and binary ** is an integer expression.

(e) A function call on one of the integer predicates LT,LE, EQ,NE,GE,GT or one of the integer logic functions NOT, OR, AND, XOR, LSHIFT, RSHIFT with integer expression arguments is an integer expression.

In addition, the concatenation of any number of dedicated integer predicates with an integer (or real) expression is an integer (or real) expression.

2.6.2 FORTRAN Interface

The arguments to a FORTRAN function call in FASBOL must be either dedicated integer or real expressions, in which case a pointer to the storage location holding the value is passed to FORTRAN (normal convention), or dedicated string variables or string literals, in which case a pointer to the first storage location of the string is passed, which can be treated by the FORTRAN program as an array holding the characters in "A6" format.

When a FASBOL entry is called from FORTRAN, the arguments specified in the calling sequence are copied into the formal argument variables; this process is reversed upon exit, achieving the effect of call by name. In the case of a string argument, the amount of the array copied into the string variable is determined by the maximum character count in the ENTRY declaration, which may be smaller than the maximum count for the variable.

Care must be taken not to form a recursive loop involving the same FORTRAN program at more than one level, since FORTRAN does not

have re-entrant capabilities.

2.6.3 Use of ? and . Unary Operators

A frequent occurrence in some applications is the successive access and assignment of an indirect variable or array element, as in

```
$X = $X + 1 ; A<25> = F(A<25>)
```

A more efficient practice is to first generate the name of the variable, using only one symbol table lookup or array mapping, and then use an indirect reference on the name for all future access or assignments on the variable. In the above cases, a more efficient way would be

```
Z = .$X ; $Z = $Z + 1
Z = .A<25> ; $Z = F($Z)
```

Field references are sufficiently efficient that the above technique does not pay in their case.

The name of an array element or field is only useful as long as the array or instance of a programmer-defined datatype exists; attempts to access or assign using the name afterwards will have unpredictable results.

Another frequent occurrence is the concatenation of null-valued functions for their sequential effects and/or succeed/fail potential. The compiler never generates code for concatenating elements that it knows will have null values, such as predicates and other null-valued primitives, as well as expressions preceded by ? and \. Therefore, preceding a non-primitive function call with the ? operator has the effect of suppressing code to concatenate it.

2.6.4 Pattern Matching

Frequently a programmer wishes to write a degenerate-type statement consisting of a concatenation of elements executed for their effect, as in

```
F(A)  F(B)  F(C)  F(D)
```

This syntax, however, is parsed as a pattern match, and, though having the same effect as intended (providing the match is successful), is less efficient in both space and time. The original intent can be achieved by enclosing the series in parentheses

```
(F(A)  F(B)  F(C)  F(D))
```

although the compiler will induce that effect if it knows all the elements will have null values (such as a series of predicates).

One SNOBOL4 programming technique that is not generally good in FASBOL is the assignment of patterns to variables in order to eliminate repetition of evaluation and construction of the same pattern. Since patterns are compiled as re-entrant subroutines, no "construction" ever takes place during execution, and only non-constant elements are evaluated. Furthermore, every variable substistituted for an explicit pattern structure costs some overhead during a pattern match due to the extra stacking and pattern-recursion involved. There are cases, however, when pattern assignment is more desirable. One case is if the pattern is large and appears in several places, it can be assigned to a variable and the variable used in its place; this saves writing and produces less object code. Another case is when the pattern contains an element which requires costly

evaluation, such as a programmer-defined function call, or character class pattern primitive (SPAN, NSPAN, BREAK, BREAKX, ANY, NOTANY) with a non-literal argument.

A generally applicable rule for all of FASBOL programming is that it is more efficient to use, wherever possible, literals instead of variables with constant value. This is especially true in pattern expressions, both for literals appearing as pattern elements and as arguments to pattern primitives. For example,

```
    ALPHAB = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ; COL = 74
    KEY = 'IF'
LOOP  INPUT  BREAK(ALPHAB)  RPOS(COL)  KEY          :S(LOOP)
```

is in every way less efficient than

```
LOOP  INPUT  BREAK('ABCDEFGHIJKLMNOPQRSTUVWXYZ')  RPOS(74)
.     'IF'                                        :S(LOOP)
```

In cases where integer constants cannot be used, it is still helpful to use dedicated integer variables, which are also ideally suited to be the objects of unary @ (cursor position assignment).

The primitive STRVAL is specifically designed to optimize the appearance of non-literal strings in patterns. One of its uses is to declare to the compiler that its argument is a string and need not generate code to cover the possibility of its being a pattern. Another use is to induce string concatenation within a pattern expression, where otherwise pattern concatenation would be compiled. For instance, if B, C, and D hold strings

```
PAT = A | B C D
```

causes three consecutive pattern/string matches to be executed for the second alternative of PAT whenever it is executed (as part of some other pattern). On the other hand,

```
PAT  =  A  |  STRVAL(B  C  D)
```

compiles into only one simple string match for the second alternative; the string concatenation of the values of B, C, and D is formed when PAT is assigned, and it is this value which is matched during execution of a pattern involving PAT.

Since FASBOL does not employ the QUICKSCAN heuristic of assuming at least one character for unevaluated expressions, left-recursive patterns will produce infinite loops at execution time, as they would in SNOBOL4 under FULLSCAN. Usually a set of patterns involving left recursion can be re-written to eliminate it; for instance, the simple case involving a pattern like

```
P  =  *P  'Z'  |  'Y'
```

that matches strings of the form 'Y', 'YZ', 'YZZ'. 'YZZZ', etc. could be re-written as the pair of patterns

```
P1  =  'Z'  *P1  |  ''
P   =  'Y'   P1
```

Finally, it should be noted that, as in SNOBOL4, changing the value of &FULLSCAN during a match may lead to unpredictable results. A similar caution applies to re-assignment of PATTERN variables during a match in which they themselves are used as unevaluated expressions.

2.6.5 Timing and Storage Management

The TIMER option permits the programmer to monitor the operation of any (or all) separately-compiled program(s), and provides feedback on where the time is being spent. It is usually the case that some small sections of code account for a large percentage of the execution time; the programmer's time is spent most efficiently optimizing those areas and ignoring the rest. Of course, after a series of optimizations, a new "bottleneck" will develop, which in turn can be measured and optimized, and the process can be iterated until the law of diminishing returns takes hold.

The QUICKSTORE option should be used by all programs that have a reasonable expectation of terminating before using up all available free storage. This case is roughly equivalent to an interpretive execution which never performs a garbage collection, except that much more storage is available to the FASBOL program because of the small run-time library.

When using the normal free storage mechanism, it is helpful to structure arrays and programmer-defined datatypes taking into account the fact that free storage blocks are dispensed only in sizes of powers of two. Of that amount, two storage locations are consumed in overhead, making 2, 6, 14 ... $(2^m - 2)$ the ideal number of array elements or fields for a datatype, and a number one greater than these results in the greatest waste of space. The primitives COLLECT and REMCOR provide the user with powerful tools to guide and monitor the free storage mechanism. Consider, for instance, a situation where many lists of items are being built over a period of time

directly related to the amount of data input; such would be the case for a program producing a cross-reference listing of some text. An efficient storage management technique which minimizes overhead in this case would be to first call COLLECT with a large block size and construct the lists as chains of datatypes. Storage fragmentation will be high and the overhead for adding an item to a list will be low. Periodic calls to REMCOR would monitor the amount of space left, and when it falls below a fixed amount, a consolidation strategy is employed, as follows. COLLECT is called with a small block size to ensure maximum recovery of storage in contiguous blocks. The larger (or all) data chains are turned into strings (which occupy less space), in a single concatenation for each chain. The process is then ready to start over; if it never reaches the consolidation stage, it is simply making maximum use of the available storage.

A certain amount of storage may be wasted by circular data structures unless the programmer explicitly breaks the ring when the structure is no longer needed. For example, the sequence

```
DATA('NODE(FATHER,MOTHER,BROTHER,SISTER)')
OEDIPUS = NODE('JOE','JANE','DICK','SIS')
FATHER(OEDIPUS) = OEDIPUS
                :
OEDIPUS = 'DEAD'
```

will result in the datatype (presumably) no longer being used, but not released to free storage either (since it still has a use count of 1 due to its circularity in the "FATHER" field). This problem is easily solved by the statement

```
FATHER(OEDIPUS) =
```

immediately preceding the last statement above.

2.6.6 Global and Local Definitions

Generally all names used within a given FASBOL compilation are local to that program. A name becomes global (at execution time) when it is declared EXTERNAL or ENTRY, or when it is entered in the run-time symbol table. For example, if program "A" wishes to access a natural variable in program "B", the former should contain an EXTERNAL and the latter an ENTRY declaration for the variable; either program can then reference it by its identifier.name. Another, less efficient, way would be for program "B" to declare the variable INDIRECT and program "A" to reference it via indirection ($); the only advantage in this case would be to prevent the name from being externalized, should there be a naming conflict with some other external entity.

Datatype names are also global, although not via the run-time symbol table. The reason for this is to allow a program to operate upon a programmer-defined datatype generated by a separately-compiled program. Each program must still execute its own DATA primitive call for the datatype definition before generating or accessing one. The names of the fields need not be the same for both programs, but their number must be the same, and they will be equivalent by their respective ordering.

CHAPTER 3

The FASBOL Runtime System

3.1 Introductory Remarks

FASBOL run-time operation is a blend of compiled code and library routines. The compiled code provides the structure of the program and builds elementary components such as elements of concatenations and patterns, out of specific instances of symbols in the program. The library routines handle the general operations which are applied repetitively to each particular set of elements in the structure. To give an example, the (string) concatenation of N elements is compiled as a structure consisting of the consecutive evaluation of the N elements, interspersed with N-1 calls on the expression-stack pushing library routine, and followed by a call on the concatenating library routine (with N as a parameter).

The execution of expressions and statements also include communication with a set of system variables and stacks (one of which is the expression-stack), in order to control inter-statement execution, depth of recursion in functions and pattern matches, and other global bookkeeping such as dynamic storage allocation (free storage), run-time symbol table, or optional timing.

It is difficult, because of their interdependence, to introduce all the concepts in a bottom-up fashion without some circularity of explanations; however, the most interesting (and involved) items, mainly how patterns and free storage work, should come after most of the operating details are clear.

3.2 Initialization, Termination, and Interstatement Operation

The system initialization routine is called immediately upon beginning execution of the main program, and takes care of the initial setting of the system registers and variables, as well as the initialization of free storage. The operating system parameter tables are accessed to determine the amount and location of core storage not used by the program, and this storage is given over for use to the free storage routines.

When an END statement in the main program or any subroutine is executed, control passes to the system exit routine which prints out the execution statistics and returns to the operating system. If an execution error occurs ("illegal data type", "call of undefined function," etc.), control passes to the system error routine which prints out information about the error and where it occurred, and makes an error return to the operating system. To aid in debugging, if the &ABEND keyword is on during a normal or error exit, a core dump of the free storage areas is done. This can be supplemented by the user by including a PMD (post-mortem dump) card to call for the post-execution dump of the program area.

Each statement is a self-contained unit of operation from which control passes internally to and from the library routines; the only time when control passes, during execution of the statement,to any other non-library code is for function calls, and pattern structures not contained in the statement. A statement is executed either by control passing to it in sequence or by some other statement executing a goto to the label attached to the statement (or if it is the first

statement of a programmer-defined subroutine, when that subroutine is called). After execution of a statement, control may pass sequentially to the next, transfer unconditionally or conditionally to some other statement via its label, or exit from a programmer defined subroutine via RETURN or FRETURN (NRETURN is equivalent to RETURN).

3.3 System conventions, Stacks, and Variables

The run-time system maintains some global variables and stacks to coordinate the various steps of executing statements and levels of recursion, both in functions and patterns.

PP, the parameter block pointer, indicates the location of a block which contains the program name for identification purposes, and also (if timing has been specified) a timing block pointer.

FE, the failpoint exit pointer, is used to hold the address of the current expression failpoint. For example, when entering a statement which contains some general expression (but not dedicated, see below) which may fail, FE is loaded with a pointer to the address to which control should be transferred in case the statement fails (which depends on the goto field). FE also contains an indication of the type of failpoint it is; when entering a negation (unary ¬ [\ on the 1108]), the current value of FE is saved on a stack and FE is reloaded to point to the local failpoint for the negation. When the negation terminates, FE is unstacked and the reverse logical result of the negation is done (i.e., fail if it succeeded, continue if it failed).

EA, the expression accumulator, is sometimes used to hold the current expression value, when the fast register which normally is

assigned this task is overwritten.

AP, the assignment pointer, is used to hold the NAME datatype of the variable in any nondedicated assignment (including immediate and conditional value assignments and cursor position assignments). For example, in a simple assignment statement, AP is loaded with the evaluated name of the variable at the beginning of the statement.

FLF is the fail flag and is used by the RETURN and FRETURN library entries to determine the course of action after achieving the previous level of recursion.

RST is the RESTART indicator which serves to indicate to a nested (i.e., independent) pattern that it is being either started or restarted during a pattern match (more details later).

SJ, CC (index register), CW(index register), RC, CF (register), serve to hold parameters for an active pattern match. SJ contains the string descriptor for the subject of the match (the string that the pattern is being matched against), as well as both the final position in the subject at successful match completion and the initial position in the subject (which may be other than before the first character if in unanchored mode). CC (current character) and CW (current word) serve to point to the current position in the subject being matched (the 1108 is not a byte-oriented machine and two index registers plus several execute tables must be employed to provide efficient character-by-character access to strings). Together with RC (remaining characters in subject), CC and CW specify cursor position and are the parameters which are saved and restored to reflect different restart positions during the match. CF is the character fail

indicator which is set when the match runs out of characters in the subject.

DT (index register) is active during either a pattern evaluation or a pattern match, and holds a pointer to a data block corresponding to an instance of the pattern structure currently being executed (the data segment of the re-entrant pattern).

TM is used to hold various parameters for the timing option when active.

SL (index register) is the (system) return link generally used for library calls and function calls. In order to minimize the saving and restoring of return links and other volatile registers, the run-time system is organized into a heirarchy of volatile register levels, with the lowest level having the smallest complement of registers, and each higher level including all the registers from the previous level plus three more (one index, one arithmetic, and one r-register). The highest level is the one governed by SL as a return link, and is the one that the compiled FASBOL program operates on. There are five stacks used by the run-time system, and although different in purpose, they all share the same design. A stack is represented by a series of blocks of storage chained together with forward and backward pointers. The block size for each stack is individually parametrized, and the current position of the stack is held in a variable; the stack position is specified by a pointer to the base of the current stack block plus an index within the block (which is held in an index register when the stack is active). When a given stack block overflows, the stack position is moved on to the

next block in the chain, or a new block is acquired from free storage for this purpose. Once the stack is popped (unstacked) beyond any block, the chain is left untouched on the (purely heuristic) assumption that once a stack achieves a certain general depth, it is likely to come at least close to that depth again. Thus the only physical limit on any stack depth is available free storage.

The five stacks are SS, ES, AS, CV, and RS, representing respectively, the system, expression, assignment, conditional value, and release stacks. SS uses SX as an index register (always active), ES, AS, and CV share EX as an index register (although usually controlled by and active for ES), and RS uses a volatile index. It is also efficient to maintain backup (or base position) values for the last four stacks (in ESP, ASP, CVP, and RSP, respectively). The reason for these backup values is to be able to easily reset the position of a stack (without methodically unstacking) under conditions which have not returned it to its initial state. Such is the case, for instance, when failure inside an arbitrary expression, which may have pushed ES to any depth,causes a jump to the failure routine; all this routine does (after determining that it is a global fail by checking FE) is reset ES from ESP and jump to the location pointed to by FE. SS has no base since it is used to hold levels of recursion and/or stacking and is always in control of everything else. When a function call (programmer-defined) occurs in an expression, for instance, a new base must be established for ES at the deeper function level so that the intermediate results on ES from the expression will not be disturbed by anything that occurs in any statements executed

at any further level of recursion. Therefore, one of the actions taken when entering the new function (and reversed on exit) is to save ESP on SS, and update ESP to conform to the current value of ES, thus providing a new base value for the new function level. These base values have no connection whatsoever with the stack block bases, which use a completely independent mechanism which is invisible to the routines which use the stacks. Thus SS can be thought of as a stack-of stacks, although it is used to hold other things, too.

ES is used within expressions and patterns, AS and CV are used only by pattern matches, and RS is used by the free storage mechanism.

3.4 Statement Types, Statement Execution, and Gotos

Statements can be either dedicated or not. A dedicated statement can be either degenerate (see syntax, Appendix 1), consisting of a dedicated real or integer expression, or an assignment of a dedicated expression to a dedicated variable of the same type (including one dedicated string assigned to another). When a dedicated expression fails, it jumps directly to the statement failpoint instead of to the failpoint routine. This is also true of dedicated expressions, inside a non-dedicated statement, that are on the top level of evaluation, i.e., are not nested inside non-dedicated expressions. A dedicated statement normally begins with execution of the library statement accounting routine, which updates &LASTNO with &STNO, update &STNO with the statement number, increments &STCOUNT (checking it against &STLIMIT and causing error termination if it exceeds the limit), and does timing accounting on the previous statement if timing is in effect for the routine. If the "NO.STNO" option is used in

compilation, the routine is not called and no accounting is done.

Every non-dedicated statement begins with execution of a library routine that does the above statement-accounting (but bypasses it in "NO.STNO" mode), sets the failpoint, if any, into FE, does the (optional) timing accounting, and checks the free storage release flag. This flag is set by the free storage package (and sometimes by a returning function call, see below) when a free storage request causes the acquisition or splitting of a block greater than some specified size (parametrized and modifiable at execution time by the COLLECT() primitive). If the flag is on, it is reset and a routine is called to release the blocks currently on RS for the given function level delimited by RSP, (see free storage). Within the statement, a failure will jump to the failpoint routine which (if it is a global failure) will increment &STFCOUNT and transfer control to the statement failpoint. If the entire statement succeeds, control passes sequentially into an unconditional or success goto, or into the next statement.

ES and ESP start out equal at the beginning of each statement, with EX containing the index for ES. This is the consequence of the previously executed statement terminating successfully or the failpoint routine resetting ES from ESP due to failure of the previously executed statement. Since a goto expression cannot fail, this will also be the case following execution of any type of goto which is not just a simple label.

RS and RSP are regularly made equal by the free storage release which occurs whenever the flag is set. When a function is called, RSP is saved and RSP is set equal to RS; upon return, RSP is restored

to its previous value.

A goto can be a simple label or an expression to calculate a label's location via indirection. A simple label is a jump (if it is a success or unconditional goto) in sequence following the code for the body of the statement, or simply becomes the statement failpoint (if it is a failure goto). If the goto is an expression, the compiled code is located sequentially following the statement body, except if it is a failure goto, in which case the code ia allocated under an independent location counter and the statement failpoint is a jump to the beginning of the goto code. The latter is necessary to keep the failure code out of the way of the success exit sequence (i.e., avoid jumping around it).

A non-dedicated assignment can be of various types. It can be degenerate, in which case the non-dedicated expression is simply evaluated. It can be an assignment where a non-dedicated expression is evaluated, and then value stored in a dedicated variable (after checking/conversion). The four other remaining types are general assignment, pattern assignment, match and replacement.

The first two are differentiated by the fact that an explicit pattern expression appears on the right side of the equals sign in the second. In a general assignment statement, the variable is evaluated and stored in AP, then the right side (object) is evaluated and the assignment library routine is called, which also checks possible output associations and performs the output if an association is in effect. In a pattern assignment, again the variable is evaluated and stored in AP, but the object, which is a pattern

consists of 0 to N independent evaluations of pattern elements that need evaluation, with each resulting value being saved in the data block for that pattern. The compiler generates the pattern match phase of the code side-by-side with the evaluation, but under an independent location counter. Eventually it is the pattern data block (which represents one instance of the pattern and points to it) which is assigned (if all element evaluations succeed) by the assignment routine. Patterns will be treated in more detail later.

Match and replacement statements differ only in the fact that in the former case, the subject is only a value used by the match, while in the latter case the subject must be a variable whose value is used for the match and which will be assigned the value generated by the replacement after a successful match. Both cases have a structure similar to the pattern assignment for the pattern section.

3.5 Normal and Dedicated Expressions

Whenever an expression contains integer or real constants, dedicated variables, or FORTRAN function calls, the compiler attempts to generate in-line, machine instruction code for as much of the expression as possible. With real expressions, this will be the case for arithmetic operations (except **, not defined for reals [use FORTRAN library])on the above-mentioned primaries. In addition, any undedicated variable without possible input associations which is arithmetically involved with any expression that is explicitly of type real must contain a real (descriptor) value at run-time, (or a type error will occur) so the compiled code generated forces a conversion to dedicated real value and includes it in the dedicated

expression. All of the above holds true for integer expressions, except that exponentiation (**) is added to the arithmetic, and the scope of the expression is extended to cover integer predicates (LT, LE, EQ, NE, GE, GT) and integer functions (NOT, AND, OR, XOR, LSHIFT, RSHIFT) with dedicated integer arguments. So that we may consider certain frequently appearing assignment statements as being entirely dedicated, a concatenation of a series of dedicated integer predicates followed by a dedicated expression is also a dedicated expression of the type given by the non-predicate element. For instance, the statement

```
REAL1 = EQ(INTG1, INTG2) REAL2 * REAL3
```

is a dedicated real assignment.

An arithmetic operation at a given level of expression evaluation cannot be compiled as a dedicated operation if at some nested level (below it) there is a library call, which over-writes the registers used by the dedicated expressions. In such a case, the equivalent arithmetic library call is compiled instead, in consonance with the general descriptor-mode scheme described below. A failing predicate within a dedicated expression at the very top level of an expression evaluation will jump directly to the statement failpoint, whereas if it occurs at some lower level (i.e., after ES has been pushed), the failure jump will be to the failpoint routine instead (to restore ES). Dedicated expressions within descriptor-mode expressions (or assigned to non-dedicated variables) have their results transformed to descriptor mode.

Normal (also referred to as non-dedicated or descriptor-mode) expressions are those which are neither dedicated (except a converted

dedicated expression) or part of the pattern match execution section of an explicit pattern expression. Normal expressions are compiled following a uniform convention for descriptor values, stacking operations on ES, and appropriate library calls. The descriptors and associated formats are briefly described in Appendix 3. During an expressions evaluation, the most recent result of some element or operation is expected to be a descriptor, held in a known register. This descriptor can then be assigned to a variable,used by some operation, or pushed onto ES for use by some binary or N-ary operation. This technique uses ES to hold temporary results and parameters to function calls. The compiled code gives a suitable indication, whenever the expected number of parameters is variable, of the actual number supplied. Each operation or function call unstacks ES and returns ES to the state it was in prior to evaluating the first parameter; in addition, the result of the operation/function is left, available to the next level in the expression evaluation. For example, the expression:

IDENT(A,B) C F(D + E,F,G)

(with D and E unpredicted) is compiled as the following sequence: evaluate C ; push ES ; evaluate D ; push ES ; evaluate E ; call undedicated add library function ; push ES ; evaluate F ; push ES ; evaluate G ; call F (three args) ; call concatenation library function (2 args). The compiled code for evaluating a variable is usually just a load of the descriptor into the accumulator, but can also involve conversion to descriptor mode if it is dedicated, or possibly input processing if it has such an association. Note also that the

compiler realizes IDENT will have a null value and does not include it in the concatenation. If IDENT fails, it will jump to the fail-point routine. The result in the accumulator after the above evaluation will be the concatenation of the value of C with the result of the function call.

3.6 Programmer-Defined Functions

A programmer-defined function name corresponds to a storage location which contains a jump to the current definition for the function; until some DEFINE() or OPSYN() primitive is executed to define the function, a call to the function will cause a jump to an error exit (initial value of function location), except in the case of functions predefined by the "entry function" declaration. A function can be redefined by executing the original DEFINE with a new starting label, by a different DEFINE, or by OPSYN.

Associated with each DEFINE() primitive function call is a storage block called the function block, which contains a parametrized description of the function prototype. These parameters are used by the generalized function call/return library routines to perform a reference to the defined function. The execution of a DEFINE call causes the starting label to be evaluated and stored in the function block, and a jump to the function block to be stored in the function location. The execution of an OPSYN call results in the contents of one function location being stored in another. The actual parameters held in a function block are: A pointer to the program parameter block of the main program/subroutine in which the function definition is compiled; a pointer to the storage block pool for the function

(explained below); the number of formal arguments, local variables, and total variable value storage required for re-entrance; a starting label (pointer); a name descriptor for each formal argument, each local variable, and for the variable corresponding to the function name (function variable). A function storage block is a dynamically-acquired block of storage used to hold values and system parameters saved when the function is called (in order to provide re-entrance). When a function returns, the block is chained into its function storage block pool and not released to free storage; the same heuristic applies here as in stack blocks. The storage block is a more efficient way of saving re-entrant information than pushing the items individually onto a stack; only one item (the storage block pointer) need be pushed onto the stack

When a programmer-defined function call is executed, the following sequence occurs:

1. Null values are pushed onto ES for missing arguments, or extra values (beyond the required arguments) are unstacked from ES and ignored.

2. A function storage block is acquired from the pool (or from free storage).

3. The function block pointer is saved in the storage block and the storage block pointer is pushed onto SS.

4. SL (the function return link), DT, FE, AP, ESP, RSP, TM+1, &STNO, and PP are saved in the storage block.

5. For the function variable, and for each local variable:

(a) The current value is saved in the storage block; for undedicated, integer, real, and pattern variables, one storage location is required; for dedicated string variables, the maximum size attainable by that string is required.

(b) A null value is assigned.

(c) Possible output associations are performed.

6. For each formal argument:

(a) The current value is saved(as in 5a)

(b) The new value is popped off ES.

(c) For undedicated variables, the use count (see free storage) of the new value is incremented and the value is stored in the variable (checking possible output associations). For dedicated pattern variables, the value is first converted to a pattern, if necessary, before doing the above. For other dedicated variables, the value is converted to the appropriate dedicated form and stored in the variable location.

7. Vanishment insurance (see free storage) is performed on all the values on ES for the current level (from ES current to ESP) and ESP is set to the current value of ES.

8. RSP is set to the current value of RS, &FNCLEVEL is incremented, PP is set to point to the new program parameter block (will be the same as the old one unless control has passed to a separately-compiled program), &STNO is set to zero, and control is transferred to the starting label.

When a jump to RETURN, NRETURN, or FRETURN is executed, the following sequence occurs:

1. A flag (FLF) is set to note success or failure.

2. PP is used to check if timing was active in the program in which the return occurred. If it was, timing is completed on the last statement executed.

3. The storage block pointer is popped off SS.

4. The function block pointer is restored from the storage block and the storage block is chained back into the storage block pool.

5. The value of the function variable is saved in EA; it is first converted to descriptor mode if it is dedicated.

6. For each formal argument, for each local variable, and for the function variable:

   (a) If the variable is undedicated, or dedicated pattern, the current value has its use count decremented and the old value is restored from the storage block.

   (b) If the variable is dedicated integer, real, or string, the old value is restored from the storage block.

7. &STNO, PP, TM+1, RSP, ESP, AP, FE, SL, and DT are restored from the storage block. If there is excessive released block accumulation (RS current is more than two stack blocks away from RSP), the release flag is set.

8. The proper string is stored in &RTNTYPE, &FNCLEVEL is decremented, and, if FLF indicates success, the accumulator is loaded

with the function value from EA and control is returned to the original calling sequence. Otherwise (failure) control goes to the failpoint routine.

A function call used as a variable (appearing on the left of an assignment) must return a value of type NAME (i.e., using the . operator), and will be checked for such before being used in an assignment; NRETURN, required by the interpreter for such a case, is unnecessary in FASBOL.

A DATA() primitive function call also gives rise to a function definition for the datatype name, fully compatible with redefinition by other function-defining primitives. When a function has been most recently defined by DATA(), however, a function call of it during execution transfers to a library routine which builds the appropriate datatype out of any arguments supplied (null if not supplied) and returns it as the function result. Field references, although identical in appearance to function calls, are compiled in a special way which causes the execution of a library routine to provide a pointer to the field within the datatype (provided as an argument to the field reference). This routine must also assure that the field is appropriate to the datatype provided and/or provide one of a set of field positions in the case of a field name used in more than one datatype. The compiled code converts the field pointer to a NAME descriptor, if a variable is required (i.e., field on left side of assignment), or fetches the value pointed to.

3.7 Free Storage Operation and Usage

The run-time system employs a set of conventions which enable

the executing program to acquire storage for a value dynamically, retain the storage block as long as it is needed, and have it released for re-use or merged with a contiguous block when no longer needed. Traditionally, there have been two basic approaches to this dynamic storage allocation problem (see Knuth[9]), the first referred to here as "Garbage Collection" and the second as "Use Count". In the garbage collection scheme, the available storage is continuously depleted, irrespective of the fact that previously acquired blocks become available for a re-use, until only a small amount remains; at this point a "Garbage Collection" takes place wherein all the active blocks in storage are marked, and then all remaining storage is made available for re-use. Besides some inherent problems involving storage fragmentation and garbage collection stack depth, which will not be discussed here, the reason this scheme is not used by FASBOL is that it requires global knowledge, at execution time, of the location of every variable that might contain a value residing in free storage. One advantage of the garbage collection scheme is that, until a collection is required, the overhead for using the scheme is low.

The "Use Count" method, in contrast, constantly monitors the availability of storage blocks by maintaining a use count for each block, which depends on the number of places it is currently being used in the program. When the use count goes to zero, it is again made available to hold new values, or merged with a contiguous block to form a larger available block. This general method is used by FASBOL, but there are some significant differences in detail, of an original nature, which make a considerable difference

in the efficiency of operation and overhead consumed by the free storage mechanism.

To begin with, the compiler permits a QUICKSTORE option for eliminating virtually all free storage overhead in programs that do not expect to consume all available storage (equivalent to the case, using the garbage collection scheme, where no collection ever takes place). Of course, no recovering is possible should there be an overflow, but the point is that if there is an overflow, there would also have been a garbage collection,putting the overhead again on a par with the modified "Use Count" system described next.

At least two key factors enter into the amount of overhead required by a use count system; the first is the method by which available storage is monitored and recombined, and the second is the method by which the actual "Use Count" mechanism of determing the availability of a block is implemented. These factors are covered in the following two sections.

3.7.1 Modified Buddy System Algorithms

The method for dispensing storage used by FASBOL involves two separate mechanisms, suggested in part by the separation of unused storage into two noncontiguous areas by the 1108 EXEC II operating system, and illustrated in Figure 1. The area in Bank 1 is allocated as "non-returnable" storage, which means that when a block is requested from this area, there is no expectation of it being returned. A pointer is maintained to the next available location; after a request for a block of size N, the pointer

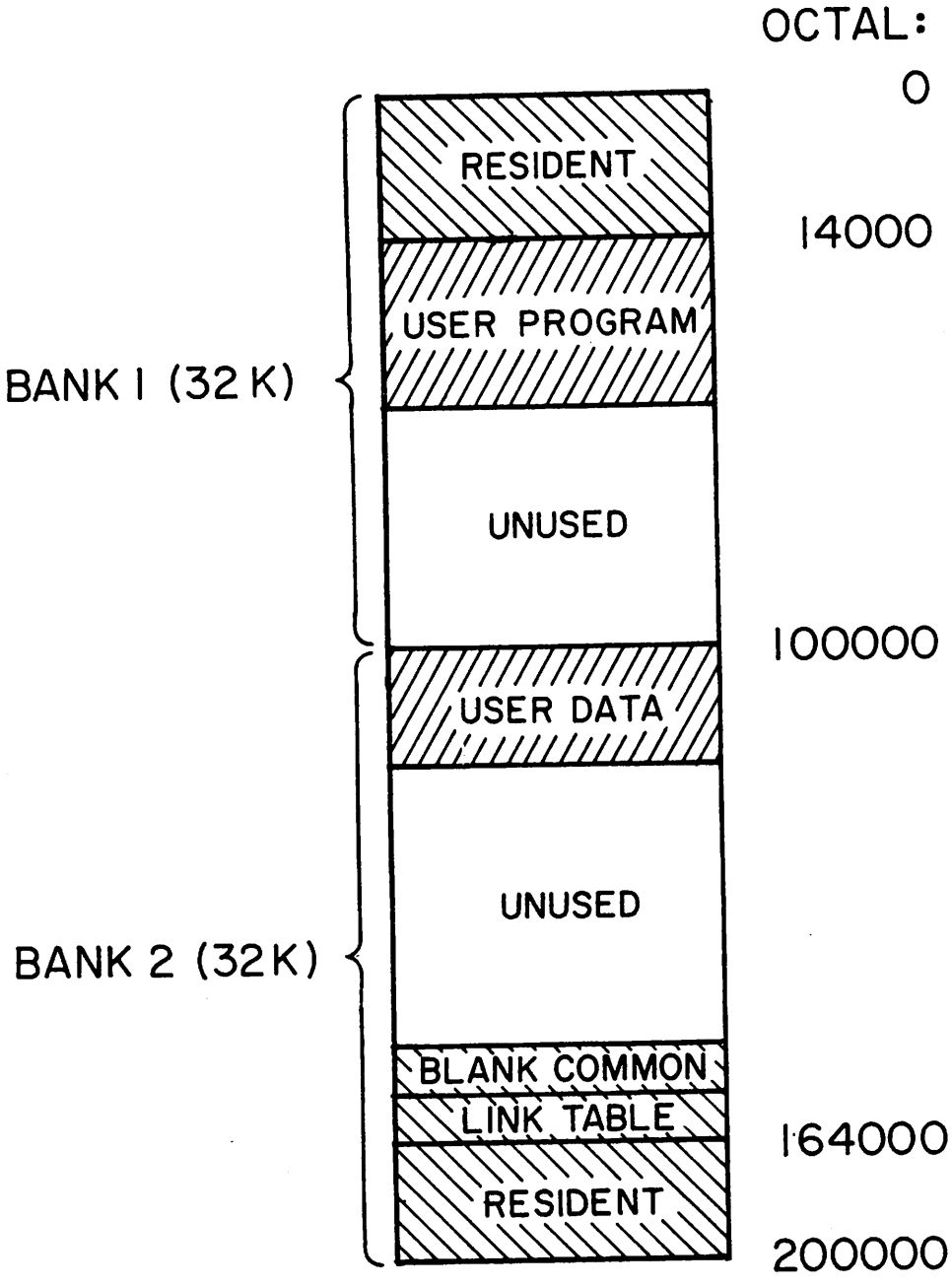


Figure 1

Storage Diagram of User Programs
Under EXEC II on UNIVAC 1108

is incremented by N. This is a suitable way of acquiring blocks for stacks, function storage, etc., which in fact are never returned. The area in Bank 2 is allocated as "Returnable", using a modified version of the "Buddy System"[10]; this system, originally used by the L^6 (L-six) language, has a distinct advantage in speed of operation over other system, with a small disadvantage in space utilization. It eliminates any possibility of long searches down a list of available blocks for one of the right size, and it assures quick and optimal coalescing of contiguous blocks. However, blocks are required to be of size 2^K, so that some space is wasted whenever a block of size between 2^{K-1} and 2^K is requested. The method described by Knuth is to maintain a separate list of available blocks of each size 2^K, $0 < K < M$. The available memory for allocation consists of 2^M words; starting at a location which is a multiple of 2^M.When a block of 2^K words is desired, and none of this size are available, a larger available block(initially 2^M) is split into two equal parts; ultimately, a block of the right size 2^K will appear. When one block splits into two, these two blocks are called buddies. If at some later time both buddies are available, they merge back into a single block of twice their size, and the process can then repeat itself. The important thing to note is that given the address of a block and its size, the address of its buddy can be easily computed as the exclusive-or of the two, since the address of a block size 2^K is a multiple of 2^K. A tag field is employed to indicate when a block is in use and when it is available, and available blocks are linked together,

by forward and backward links of a doubly linked list, to the appropriate list head for the given size. In FASBOL, the initially available storage is not a power of two, nor does it start on such a convenient boundary; furthermore, the free storage system, which uses the Bank 2 area for returnable storage initially, will convert the remainder of the Bank 1 area to the buddy system if the former overflows. Therefore, the FASBOL free storage system uses an additional algorithm to prepare any arbitrary area for use by the buddy system, which is modified to cope with the possibility that not all buddies of blocks within the area are also within the area. Furthermore, the buddy system liberation algorithm (for returning a block), as described by Knuth, has a serious deficiency in that it will only work if blocks are released in the reverse order from that in which they were acquired. It turns out that available blocks must also carry information as to their size, since a starting location for a block of size 2^K can also be the start of any smaller block, and when a block is released and its buddy computed, it is not sufficient to know only if it is available but whether, if available, the "Buddy" is also of the same size. The L^6 paper[10] does mention this parameter.

Three algorithms used by the FASBOL free storage system are given below, corresponding to initializing an area of storage for use, acquiring a block, and returning a block. The free storage system maintains an array of entries AVAIL(1) through AVAIL(M) which serve as heads of the list of available blocks for the corresponding power of two. The fields LINKF and LINKB of these entries, and

of the first word of an unused block, are the forward and backward chain pointers for the lists. Defining LOC(A) to be the address of item A, an empty list for size K is indicated by LINKF(AVAIL(K)) = LINKB(AVAIL(K)) = LOC(AVAIL(K)). The field TAG of a block is 1 if the block is in use and 0 otherwise; the field SIZE of a block is the size (1 to M) of the block. KMAX is the largest block size in use by the system (largest found during any initialization). In the acquisition algorithm given below, when TK exceeds the release block size, the release flag is set.

Initialization Algorithm

Let UB,LB be the upper and lower bounds of the storage area to be initialized.

Let SADR = BLKS = 2^M , K = M

L1 (Find window in storage)

If K = 0 , OVERFLOW (no space available)

If SADR > LB, go to L2

BLKS ← BLKS/2 ; K ← K - 1 ; SADR ← SADR + BLKS

Go to L1

L2 If SADR < UB , go to L3

BLKS ← BLKS/2 ; K ← K - 1 ; SADR ← SADR - BLKS

Go to L1

L3 KMAX ← MAX(KMAX,K)

SAVE SADR,BLKS,K

L4 (Staircase UP)

If K = 0, go to L5

If SADR + BLKS < UB, BEGIN

Call ADD(SADR,K) ; SADR ← SADR + BLKS END

BLKS ← BLKS/2 ; K ← K - 1

Go to L4

L5 RESTORE SADR,BLKS,K

L6 (Staircase DOWN)

If K = 0, FINISHED

If SADR - BLKS > LB , BEGIN SADR ← SADR - BLKS

Call ADD(SADR,K) END

BLKS ← BLKS/2 ; K ← K - 1

Go to L6

ADD Subroutine

ADD(SADR,K) ADD BLOCK to AVAIL list

LINKF(SADR) ← LINKF(AVAIL(K))

LINKB(SADR) ← LOC(AVAIL(K))

TAG(SADR) ← 0

SIZE(SADR) ← K

LINKB(LINKF(SADR)) ← LOC(SADR)

LINKF(AVAIL(K)) ← LOC(SADR)

RETURN

Acquisition Algorithm

Let K = power of 2 such that 2^{k-1} < required block size < 2^{K}

TK ← K

L1 (Search for available block)

If TK > KMAX , OVERFLOW

SADR ← LINKF(AVAIL(TK))

```
        If   SADR ≠ LOC(AVAIL(TK)) ,  go to L2
        TK ← TK + 1
        Go to L1
L2      LINKF(AVAIL(TK)) ← LINKF(SADR)
        LINKB(LINKF(SADR)) ← LOC(AVAIL(TK))
L3      (Look for right size)
        If   TK ≠ K , BEGIN TK ← TK - 1 ; go to L4 END
        TAG(SADR) ← 1
        SIZE(SADR) ← TK
        FINISHED (with address of acquired block in SADR)
L4      (Split Block)
        TADR ← LOC(SADR) + 2^TK
        CALL  ADD(TADR,TK)
        Go to L3
```

Return Algorithm

```
        Let   SADR  =  LOC of block being returned
        Let   BOUNDS = {(LB1,UB1) , (LB2,UB2), ..., (LBN,UBN)}
```

i.e.,the set of N areas allocated to the Buddy System.

```
           (merge loop)
L1      K ← SIZE(SADR)
        If   K = KMAX , go to L5
        TADR ← XOR(SADR,2^K)
        I    ← 1
L2      (Bounds Loop)
        (LB,UB) ← BOUNDS(I)
        If   LB < TADR < UB , go to L4
```

```
L3   If   I = N , go to L5
     I ← I + 1
     Go to L2
L4   If   TAG(TADR) = 1 , go to L5
     If   SIZE(TADR) ≠ K , go to L5
     If   TADR < SADR , SADR ← TADR
     LINKB(LINKF(TADR)) ← LINKB(TADR)
     LINKF(LINKB(TADR)  ← LINKF(TADR)
     K ← K + 1
     Go to L1
L5   CALL  ADD(SADR,K)
     FINISHED
```

3.7.2 Use Count Algorithm

The algorithms given in the previous section are part of a larger mechanism which provides for the creation, maintenance, and release of values requiring dynamic storage. Each free storage block that is in use has a use count field and an RS flag field, the former being an indicator of its required existence and the latter an indicator of whether the block (descriptor) is on RS (0 if not on, 1 if on the release stack). The general philosophy is to increase the use count of a block when the value it represents is used in a new place in the program, such as being assigned to a variable. When a value is replaced by another or discarded, the use count is decremented; what actually happens is that if it is not on RS already, it is put on RS, and otherwise the use count

is decremented, since being on RS is equivalent to one less use. A series of policies is now described which together form an algorithm for both minimizing the manipulation of use counts and effecting an automatic and efficient return of unused blocks (to the third algorithm of the previous section). Since a block is not actually merged back into free storage until the release stack is purged, this algorithm has the effect of concentrating most of the overhead into a series of pseudo-garbage collections; by delaying the re-merging of buddys, the situations where the same block(s) are repeatedly broken apart and then merged back together are minimized.

(a) Any storage value that results from the evaluation of any part of an expression does not have its use count incremented. For example, when the value of a variable is used, it is simply loaded into the accumulator. Even though it represents a fresh use of the value, the use is considered an intermediate result.

(b) When a library routine is forced to acquire a free storage block to hold some newly created value, such as the result of a concatenation, the block is given a use count of 1 but is also put on RS, thus having an effective use count of 0.

(c) When a new value is assigned to a variable, the use count of the old value is decremented and the use count of the new variable is incremented. This applies not only to explicit assignments, but to any situation where a new value replaces an old one, such as in elements of arrays or fields of datatypes, active input associations, etc. In some cases, there are no old values to be

replaced, such as in the creation of a new instance of a programmer-defined datatype.

(d) At each function level, there is a segment of RS which contains the accumulation of blocks that were either created or had their use count decremented at that level (and were not already on RS for some previous higher level). If, upon entering a new statement, the release flag is found set, the free storage package is called to release all the blocks on RS for the current level. This goes as follows:

1. If RS current = RSP , FINISHED.

2. POP a block off RS and decrement the use count field. If not 0, go to 1.

3. If the block corresponds to an array, data or pattern value, put each element value on RS (or decrement its use count if already there). What this means is that when a complete item such as an array ceases to be used, there is one less use for each of its elements.

4. Exercise the return algorithm on the block and go to 1.

(e) The above policies prevent the extra incrementation and subsequent decrementation of intermediate values and provide for the automatic release of unused created values. Because of the tenuous existence of intermediate values, however, an additional policy is required for the execution of programmer-defined function calls, called "vanishment insurance". For example, suppose variable "X" has a string value, and X is the only place where this string

is currently used, giving it an effective use count of 1. Furthermore, the string value has not been put on RS for any reason, and X is used in an expression where it is an element of a concatenation. Another, subsequent element of the concatenation is a function call, and within that function X is assigned a new value. This will cause its previous string value to have an effective use count of 0, so that if a release is also triggered inside the function, the storage used by the string will be reclaimed by the free storage package and re-used, despite the fact that it is still needed by the expression where the function is called. Indeed, the descriptor for the string is on ES, but the descriptor points to an area that may contain something completely different. To prevent, in general, the release of values still in use at a higher function level, the library function which performs programmer-defined function call entries, after unstacking the function's actual arguments, does a vanishment insurance on every value which is on ES between the current value and ESP. This insurance consists of putting all values which are not on RS onto RS and also incrementing their use count, values already on RS are unaffected. Although not changing the effective use count of any value, this policy insures that the values cannot go onto RS (and be returned to free storage) at some deeper function level. Eventually, a release triggered at the level of the function call will take the value off RS. This always occurs between statements, and in particular after the value has been used in the concatenation, etc., so that if its use count goes to zero, it can be returned without any problem.

3.8 Patterns and Unevaluated Expressions

Perhaps the most novel idea in FASBOL is the way in which patterns are treated. Interpreters and semi-interpreters evaluate patterns as intermediate-language data structures, and perform pattern matches by executing a generalized routing which uses the intermediate data structure as a template to compare against the subject string. FASBOL compiles each explicit pattern into two logically separate sections of code, which are generated side-by-side under separate location counters. The first section corresponds to the evaluation of the pattern, and the second is a re-entrant subroutine corresponding to the execution, during a pattern match, of an instance of the pattern. Associated with each instance of a pattern is a data block, consisting of a pointer to the pattern subroutine and the values, if any, of its elements. During pattern evaluation, any element of the pattern that has a non-literal value (such as a variable) is evaluated and its value saved in the data block. No evaluation is required for the actual structure of the pattern, which consists of the arrangement and use of concatenations, alternations, immediate and conditional value assignments, cursor position assignments, unevaluated expressions, primitive pattern variables and primitive pattern functions. When the pattern routine is executed during a pattern match, it uses any values that are in the data block as parameters for the match. For example, the pattern element ANY('0123456789') requires no code for the evaluation phase (since it has a literal argument), and generates the following code for pattern matching: Load break table; call 'ANY' library routine; jump to failpoint. A break table is a bit table

specifying a set of characters (in this case the digits 0-9) which is used by ANY to see if the next character in the subject is a member of the class, and it will skip over the failpoint jump in the code if it is successful.On the other hand, the pattern element STRVAL(X) generates code for the evaluation phase that loads the value of X and stores it in the data block; the match phase code loads the value from the data block and calls the string match routine to compare the value with the subject.

The illustration in Figure 2a. is an abstract symbol for representing the external characteristic of any pattern, either simple or complex. "START" is the entry into the pattern, "SUCCEED" is the exit after a successful match, "FAIL" is the exit after an unsuccessful match, and "RESTART" is the entry at which the pattern can be restarted after an initially successful match. The actual code generated for the pattern match execution follows the conventions of Figure 2b. "START" and "SUCCEED" are accomplished by entering and leaving the code sequentially, while "FAIL" is a jump out of the immediate pattern code to the fail point (dictated by the structure surrounding the pattern) of the immediate pattern. "RESTART", for patterns with alternatives (implicit or explicit), is a label on a line of code inside the pattern which, when executed, will restart the pattern match on a new alternative of the immediate pattern. If a pattern has no alternatives, "RESTART" is equated to "FAIL"; thus any number of patterns can be made transparent to backup, a restart going directly to the last pattern with alternatives. Figures 3 and 4 should help clarify some of the above ideas;

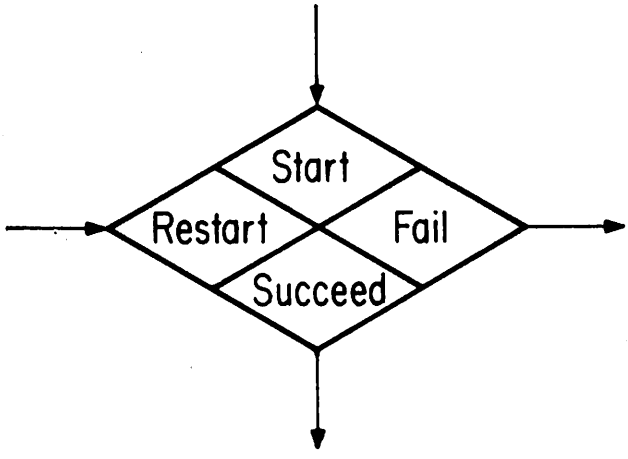


Figure 2a

Abstract Symbol for a Pattern

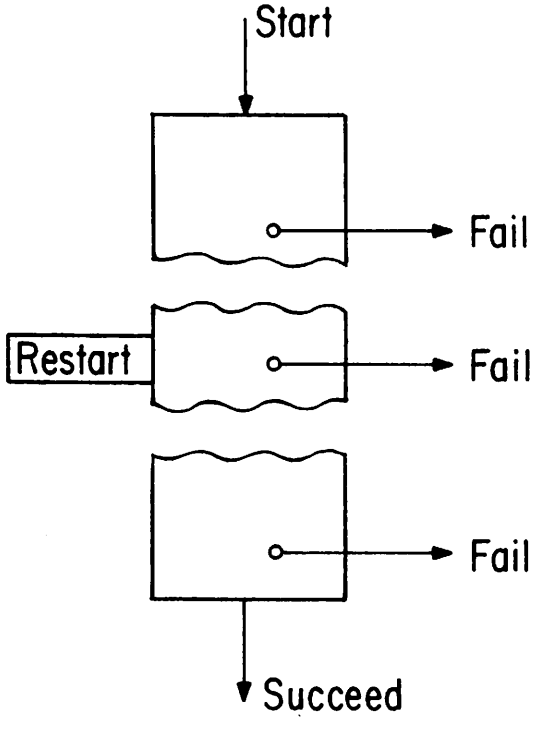


Figure 2b

Physical Appearance of a Pattern

Figure 3 is a symbolic representation of the pattern structure corresponding to the concatenation of three other patterns, and Figure 4 gives both a symbolic and a functional description of the alternation of three patterns. A pattern structure can be thus recursively described, from the outermost structure of a pattern assignment or match, down to the individual elements which are not structurally composed of others.

Pattern match execution involves two stacks, ES and AS, which operate in orthogonal fashion to each other. ES gets deeper (more items on the stack) as the match progresses successfully along the subject string, and is deepest upon successful completion of the match. It is used to save items which are required if and when the match backs up upon failure and must restart its last alternative. Such items are, for instance, the position of the cursor at each point in the subject where several alternatives could be matched, and the next restart point of a group of alternatives (see Figure 4). AS get deeper with nesting of patterns within conditional or immediate value assignments and/or re-entrant calls to other patterns compiled elsewhere; AS will be at its initial position upon successful match completion, and is used to hold return links and data block pointers when pattern subroutines are called, as well as initial cursor positions for immediate and conditional value assignments, ARBNO() parameters, etc. A third stack, stack, CV, is used to hold parameters for conditional value assignments; although it operates in general synchronism with ES, it provides a more convenient way to access any conditional value

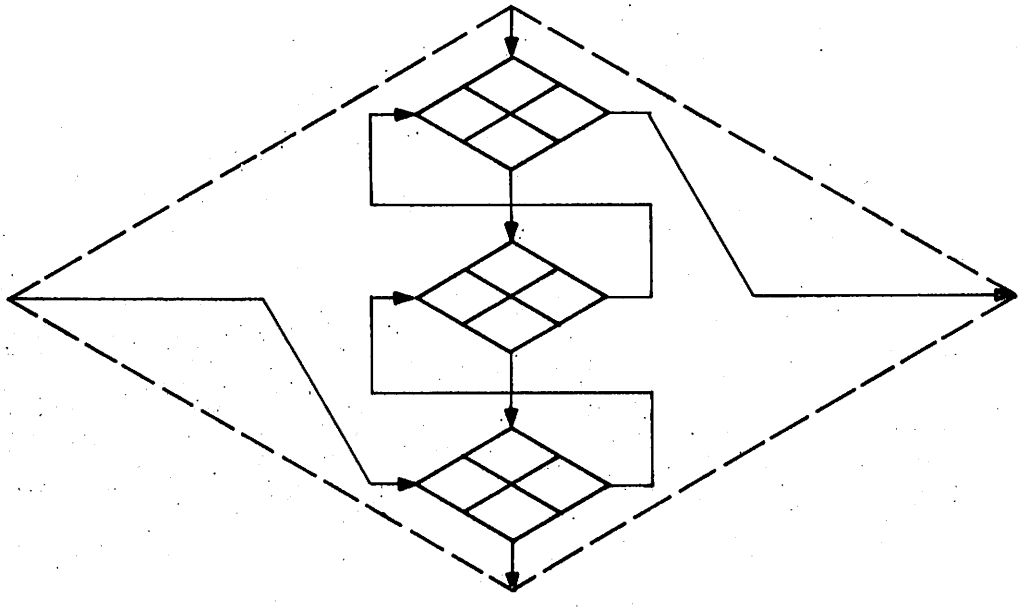


Figure 3

Concatenation of Three Patterns

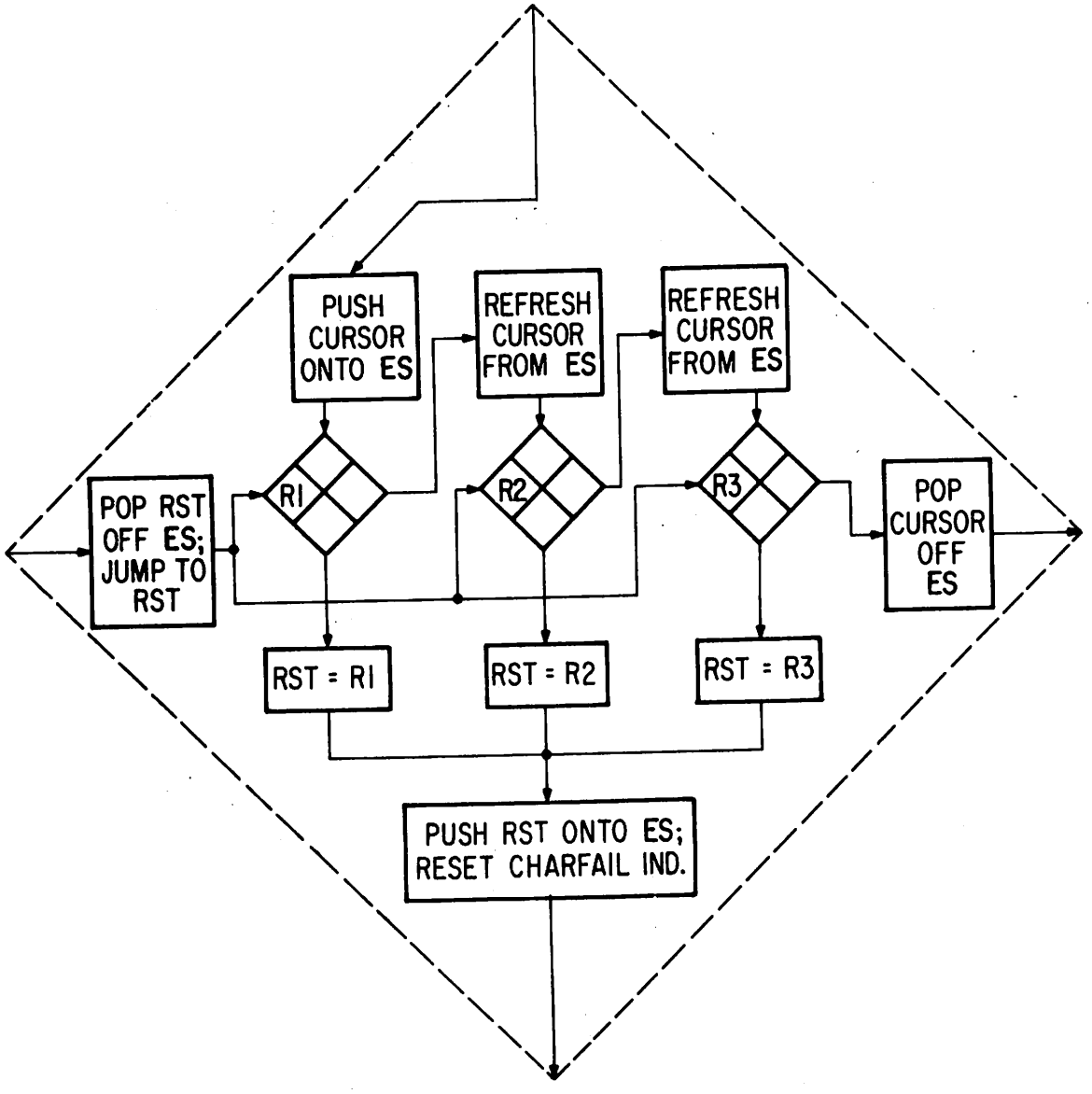


Figure 4

Alternation of Three Patterns

parameters upon successful match completion, instead of having to unstack ES looking for them. Figures 5 and 6 provide additional illustrations of how the stacks are used, in this case by primitives. Note the evaluation phase in Figure 6, due to a non-literal argument in SPAN.

The evaluation phase of a pattern assignment statement is executed as follows:

1. Evaluate the variable and save it in AP.
2. Call the 'GET PATTERN BLOCK' library routine which:
   (a) acquires and zeros a data block from free storage
   (b) Stores in it a pointer to the pattern match routine
   (c) Pushes the data block (pattern descriptor) onto ES and stores it in DT.
3. Evaluate each element of the pattern that requires evaluation, increment the use count of the resulting value, and store it in the data block.
4. Pop the pattern descriptor off ES and perform the assignment (just like other types of assignments).

The execution of a pattern assignment generates an instance of the pattern, represented by the data block, for specific values of the elements at the time of assignment. Like any other storage value, that particular instance of the pattern lasts as long as it is used in at least one place, i.e., as the value of a variable or as part of some other larger, evaluated pattern. When a pattern descriptor is no longer used and returned to free storage, the use counts of all of its evaluated elements are decremented. Repeated

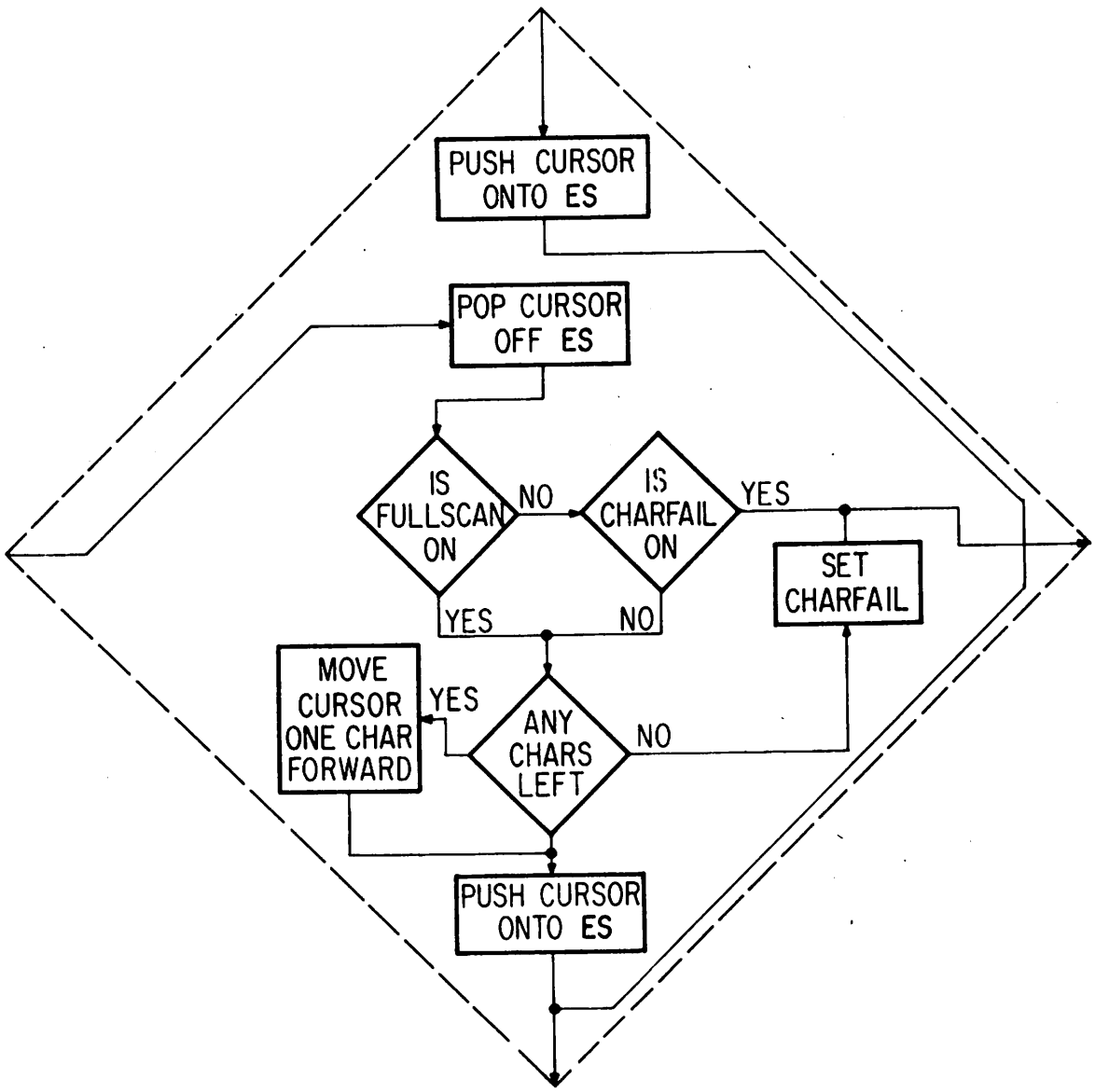
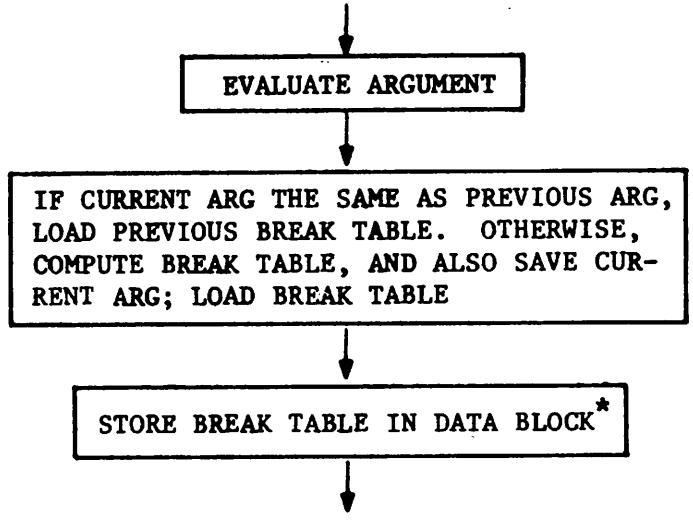


Figure 5 ARB Pattern

A. EVALUATION PHASE



*EACH PATTERN HAS ASSOCIATED WITH IT A DATA BLOCK WHICH CONTAINS THE EVALUATED PATTERN PARAMETER VALUES. THESE VALUES ARE COMPUTED DURING THE PATTERN EVALUATION PHASE AND USED DURING THE PATTERN MATCH PHASE.

B. PATTERN MATCH PHASE

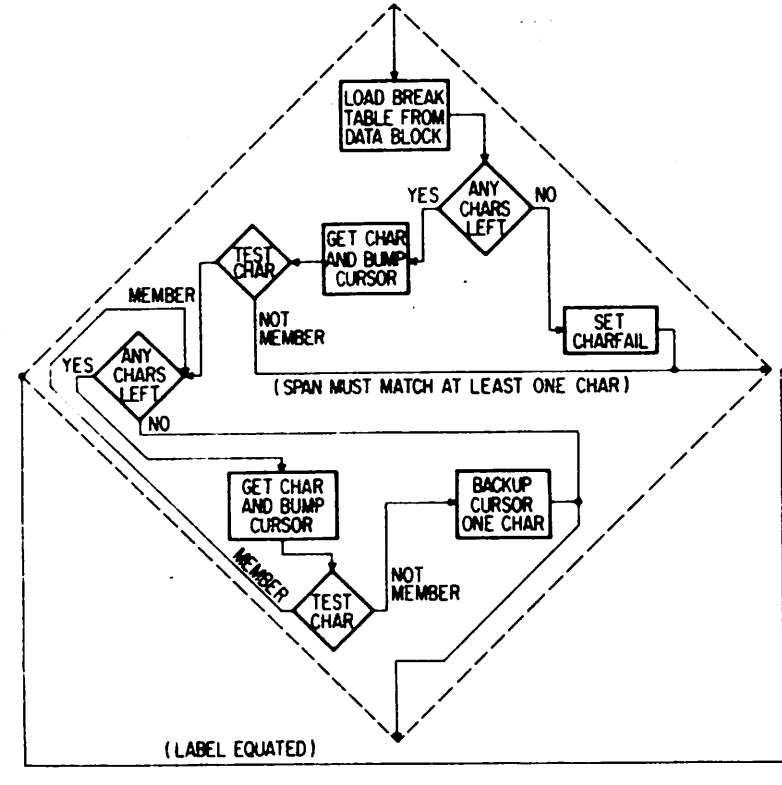


Figure 6

SPAN Pattern with Non-Literal Argument

executions of a pattern assignment generate fresh instances of the pattern.

The pattern match code generated for the assignment statement consists of the structural code described previously with the following additional conventions. The failpoint of the entire pattern subroutine is the pattern fail library entry, and a jump to the pattern succeed library entry immediately follows the pattern match code to provide for a success exit; these two entries are analogous to FRETURN and RETURN. In addition, immediately preceding the pattern match code is a jump to its restart point (which may be the pattern fail library entry, if the pattern has no implicit or explicit alternatives). It is this first storage location (jump to restart) that is pointed to by the data block for the pattern. When a pattern match is executed which uses an instance of the pattern (i.e., if a variable to which it is assigned appears as part of some other pattern expression), the following sequence occurs:

1. The other pattern sets RST to 0 if it wishes to start the inner pattern or 1 if it wishes to restart it. Pattern routines always have possible restarts, since the outer pattern has no knowledge of their nature.

2. The pattern execute library routine is called to enter the pattern subroutine; it is analogous to the function call entry routine. This routine pushes DT and the return link to the outer pattern onto AS, sets DT to the new (subroutine) data block, and jumps to either the first or second location of the subroutine

for restart or start, respectively.

3. The pattern routine executes, possibly calling other pattern routines, until failure or success transfer control back to the fail/succeed pattern library entries.

4. These entries set FLF to 1 or 0, respectively, pop DT and the return link off AS, reset DT to its previous value, and jump to the first or second location after the pattern execute call in the outer pattern, signifying, respectively, failure or success of the pattern subroutine.

The evaluation phase of a pattern match statement is executed as follows:

1. Evaluate the subject string and push it onto ES.
2. Call the 'GET MATCH BLOCK' library routine which:
   (a) Loads DT with a data block pointer which it acquires for a data block pool for the statement, or from free storage.
   (b) Saves a pointer to the block pool pointer in an internal location for possible use by the unevaluated expression routines. The block pool pointer points at the current block being used at a chain of any other previously used blocks (see unevaluated expressions).
3. Evaluate each element of the pattern that requires evaluation, but do not increment the use count before storing in the data block.

After the evaluation phase, execution passes to the match pattern library routine, which:

1. Pops the subject off ES, makes sure it is a string, makes sure it won't vanish, and pushes it back onto ES.

2. Initializes SJ and the cursor (CC, CW, RC).

3. Pushes the return link to the program (following match execution) and the current value of ES onto SS.

4. Transfers control to the first location of the pattern match execution, which is pointed to by the data block.

The pattern match code generated for the match statement has as its failpoint the failed match library entry, and jumps to the succeeded match library entry on success. Its first location is its start point, and any restart point is ignored, since the outermost pattern of a match is never restarted.

The failed match entry jumps to the abort match library entry if:

(a) &ANCHOR is nonzero, or

(b) There are no more initial positions for the subject (unanchored mode has run out of characters), or

(c) &FULLSCAN is zero and the CHARFAIL indicator is on (not enough subject for pattern).

Otherwise, CHARFAIL is reset, the initial position and the cursor are incremented, and the pattern match code is started again (NOT restarted).

The aborted match entry, which is also called if there is an abort anywhere within a pattern match (including pattern subroutines),

pops ES and the return link off SS, resets ES to its previous value (and ignores the link), restores AS and CV from ASP and CVP, and jumps to the failpoint routine (which will cause the statement to fail).

The succeeded match entry computes and stores the final position does any conditional value assignments using CV and CVP, restores AS and CV from ASP and CVP, pops ES and the return link off SS, resets ES to its previous value, and returns to the code immediately following the match pattern (library) call, which is the successful exit from the statement.

A replacement statement executes similarly to a match statement, except that the name descriptor for the variable (subject) is pushed onto ES first, before the subject itself. Upon return from the succeeded match routine, SJ is pushed onto ES, the object field is evaluated, and the replace library entry is called, which

1. Pops SJ and the subject off ES.

2. Creates a string formed from the replacement by the object, of the substring between initial and final positions of the match in the subject string.

3. Assigns the replaced string to the variable (popped off ES).

Unevaluated expressions are handled using the simple device of compiling them under the location counter for the pattern match code rather than the one for the pattern evaluation code, thus deferring the evaluation until the pattern match execution reaches that point. If the expression cannot fail, no extra code is

generated (in fact, no code is needed to load and store values in the data block). If the expression may fail, the code is preceded by a call on the unevaluated expression begin library entry, giving as parameters the failpoint of the unevaluated expression and a flag indicating if there are any programmer-defined function calls within the expression. The code is followed by a call on the unevaluated expression end library (which gets executed in the case that the unevaluated expression succeeds),and, at the location previously designated as the unevaluated expression fail point, a call on the unevaluated expression fail library entry. The parameters passed to the latter two entries are the previously mentioned function call flag, and the pattern failpoint for the expression.

If the function flag is not on, the unevaluated expression start entry just pushes ESP and FE onto SS, resets ESP to the current value of ES, and loads FE with the unevaluated expression failpoint. Upon success or failure of the unevaluated expression, the process is reversed and control is passed to the next location in sequence or to the failpoint specified by the pattern structure. If a programmer-defined function is called within the unevaluated expression, more parameters must be saved (and restored), since another pattern match may be initiated within the function. In particular, ASP, CVP, SJ, and the cursor are pushed onto SS, and ASP and CVP updated to the current values of AS and CV. Furthermore, the possibility exists that during the function execution control may pass through the controlling match for the unevaluated expression (the unevaluated expression may be

in the same statement as the match, or in some pattern subroutine). If this happens, a new data block for the controlling match will be acquired from the pool (or free storage),so that pointer to the current block pool pointer and its corresponding current data block pointer must be pushed onto SS, and the block pool pointer updated to no longer point to the current data block. Also, the element values, if any, in the current data block for the controlling match must have vanishment insurance performed on them, since their use count was not initially incremented. This entire method of dealing with unevaluated function calls allows all other pattern matches that do not use this feature to run with a minimum of stack and use count manipulation, and only penalizing those which do use the feature.

3.9 Other Run-Time Features

3.9.1 FORTRAN Interfaces

When calling FORTRAN from FASBOL, each external FORTRAN program is typed as integer or real; each argument in the call must be a dedicated integer or real expression, a string variable,or a string literal. A standard calling sequence is compiled, with the result expected in the system accumulator register on return. Strings may be considered by the FORTRAN program as arrays with characters packed six to a word. If an external FORTRAN function is declared 'OPEN' (i.e., capable of calling other FASBOL programs), the calling sequence is preceded by a library function call which saves the non-volatile registers used by the FASBOL run-time, and, is followed by a library call to restore them. In this way any

FASBOL programs called by the FORTRAN program can resume normal operation and reflect back changes in global parameters.

FASBOL can only be called from FORTRAN after passing through at least one level of FASBOL programs (MAIN must be FASBOL). The FORTRAN entry declaration serves to externalize the function entry, type the function variable as integer or real, type each formal argument as integer, real, or string (with max string size for the latter), and provide a starting label and return label for the function definition. The formal arguments are copied from their counterparts in the FORTRAN calling sequence (with the max string size copied for strings) on entry into FASBOL, and then copied back (possibly modified) on return to FORTRAN, with the function variable value left in the system accumulator register. In addition, on entry all the non-volatile registers affected by FASBOL are saved (this may be in use by FORTRAN), the previous FASBOL value of these registers are restored, and the parameters associated with the FASBOL statement which originally called the FORTRAN program (STNO, PP, LINK, TM+1, RSP, ESP, FE, AP, DT) are saved; the process is reversed on exit back to FORTRAN.

3.9.2 Symbol Table

The run-time symbol table contains entries for variables, labels, and functions. It is implemented as a chained hash table [11], with the number of buckets (and hash bit width) capable of being set by a compiler option ("HASHSIZE = N "); the default is N=6 (64 buckets). The hash scheme multiplies the first word of the symbol string by the last, and extracts a key of the appropriate

bit width from the "Normalized" (compensating for zero fill of the last word) middle of the product. The chain from each bucket is ordered in increasing symbol size; each entry on the chain consists of a link word, the symbol string descriptor, and a NAME descriptor for the variable label, or function. The link word contains a pointer to the next link and a character count and type field which permits a rapid search down the list, with string comparisons only necessary when (and if) an entry with the same character count and type is found. A search stops whenever an entry with a larger character count is encountered.

The symbol table routine operates in two modes; initalization and indirection. The initialization mode is for entering items declared to be 'INDIRECT', where both a string descriptor and name descriptor are provided and new entries created. The indirect mode is for all other symbol table references; a string descriptor and partial name descriptor are provided, the partial name descriptor indicating the type (VAR, LAB, FUN) desired, and possibly,for variables, and I/O association flag. A search is made for the symbol/type combination,and if found, the partial name descriptor and full name descriptor from the symbol entry are "OR"ed together, stored back in the entry, and provided as the result of the indirection. If the symbol is not found, a new entry is created; and a new name produced for the created entity. In the case of a created variable, an additional storage location is acquired and initialized to null; in the case of a function, an additional storage location is acquired and initialized to jump to the undefined

function error exit; and in the case of a label, the name points to the undefined label error exit.

3.9.3 Input and Output

I/O is done through I/O associations of variables (including created variables that are array elements) in the program. Whenever evaluating by value (as opposed to by name) an identifier (flagged by the compiler at having a possible input association), indirect reference, or array reference, a library routine is called which searches an input association table for that variable and performs the input if it finds it there. Whenever a program assignment is made via =, $, ., @, or formal argument binding, the name descriptor of the variable is checked for an output association possible flag. If it is on, the output association table is checked, and if the variable has a current association, its value is output.

I/O associations are made using the primitives INPUT and OUTPUT, which have three arguments: variable name, unit number , and association length. If the first argument is a literal string, the compiler generates the corresponding name descriptor for the variable; otherwise, the argument is evaluated and an indirect reference to get its name from the symbol table is performed. The unit no. refers to the standard FORTRAN logical unit designations as defined by the system or user FORTRAN unit table. The association length is the string length that FASBOL uses as its logical record length for a given association. On input, the association length must be no greater than the physical record length, and

provides a truncated record if it is less; one physical record is always read, and any excess record size is lost. On output, the following rules apply, where SL = string length, AL = association length, and RL = record length: AL > RL is an error; AL < RL causes blank fill to the full RL; SL < AL causes blank fill to the full AL; SL > AL causes multiple records of length AL with the last blank filled to AL.

The variables INPUT, OUTPUT, and PUNCH have predefined initial associations corresponding to INPUT('INPUT', 5, 80), OUTPUT('OUTPUT', 6, 132), and OUTPUT('PUNCH', 7,80) for the card reader, printer, and punch, respectively. I/O associations can be dynamically changed by using the input and output primitives, and discontinued by using the DETACH primitive. Additional I/O operations are provided by ENDFILE, REWIND, and BACKSPACE for any unit capable of such an operation, and EJECT for ejecting to top of page on the printer. The physical devices provided for in the I/O library routine are: CARD READER, PRINTER,PUNCH, CONSOLE KEYBOARD, TAPE, and DRUM.

Tape and drum operations are buffered to N levels on output and double-buffered (one record look-ahead) on input. The number of output buffer levels is a function of the record length, since the total amount of storage used for output buffering by any device is a program-controlled constant. All buffers come from a common buffer pool and are originally acquired from free storage as they are needed. The buffering scheme uses the device end-action interrupts to permit concurrent execution of the program and I/O

operations. Input look-ahead can be disabled with the "NO-LOOK-AHEAD" compiler option; this is only necessary if a FORTRAN subroutine is also performing I/O concurrently on the same logical unit.

3.9.4 Primitives

Primitives are recognized by the compiler and not treated like programmer-defined functions. This has only one disadvantage (no re-definition of primitives), and many advantages. Since the compiler knows how many arguments each primitive should have, it can compile extra null arguments; the number of actual arguments is not passed to the primitive, and it does not have to check the actual number against the expected number. The compiler knows that no primitives cause recursion, which primitives have null values, and which primitives may fail, permitting a certain amount of optimization of the generated code. Certain primitives are treated in a special way, and in some cases are compiled as in-line code. Pattern primitives are incorporated into pattern structures, only sometimes calling library routines. Integer-argument integer predicates, integer-argument integer functions, and IDENT and DIFFER with only one argument are compiled as in-line code. DEFINE, DATA, ITEM, and APPLY have their arguments treated in special ways, as do OPSYN, DETACH, INPUT, and OUTPUT, in the latter four cases,because a name descriptor is what is really needed for some arguments, even though the actual argument is usually a string. Finally, REPLACE and the character-class pattern primitives(SPAN, NSPAN, BREAK, BREAKX, ANY, NOTANY) make use of a special run-time optimization technique: some

times a special table must be built, using one or more arguments, before performing the actual operation specified by the primitive; instead of building the table each time the primitive is called, it checks to see if the arguments provided are the same as in the last instance of the call, and if so,it uses the old table. The old arguments and table require program storage in each place the primitive is called, and this must be generated by the compiler.

CHAPTER 4

The FASBOL Compiler

4.1 Structure

The FASBOL compiler is a one-pass translator which accepts statements in FASBOL and produces symbolic assembly-language output. The assembler is automatically invoked upon completion of compilation. The compiler operates as a system processor like the FORTRAN compiler and the assembler, with the same capability for file manipulation and updating. The compiler is written in FASBOL, with a few assembly-language subroutines. There are several reasons for writing it in FASBOL rather than, say, assembly language. One is speed of implementation; the chief contribution of FASBOL is in the speed of execution of the compiler programs, and the time required to write and get the compiler running was minimal (about 1 month to write, 1 1/2 months to hand-compile the first version, and 1 week to debug, mostly hand-compilation errors). Another reason is the general ease with which compiler errors can be fixed, and new versions and/or extensions implemented (for example, making FASBOL compatible with version 3 of SNOBOL4, a development that occurred too late to be included in the first version of FASBOL). A third reason was to illustrate the power of FASBOL for compiler-writing and other similar applications, and its ability to compile large programs (one phase contains over 1600 lines and over 1,000 statements, takes 45 seconds to compile, and generates over 13,000 machine instructions and data words).

The compiler is structured in storage as illustrated in Figure 7, where the main program, FASBOL, remains loaded throughout the compilation and the five phases are successfully (and automatically) loaded on top of each other as the compilation proceeds , starting with OPENSR. The main program initializes global parameters, controls the sequence of the phases, and contains the functions used by more than one phase, such as the "Get Next Statement" function. OPENSR provides the standard processor input editing capability of inserting, deleting, or replacing lines in the source code, and of creating an updated source element with the same or different name. DECLPH processes the declarations, which are required to preceed all executable statements. PROCPH processes the executable statements up to the "End" statement. EACTPH generates program storage and constants and CROSPH, which is optional,is called if the program specified a cross-reference dictionary listing. The main program invokes the assembler upon the symbolic output element (unless -NOASM has been specified).

The remaining sections discuss the compiler symbol table arrangement, and each of the phases(except OPENSR, which is a standard system program).

4.2 Symbol Table

The FASBOL compiler makes full use of the run-time symbol table to hold an entry for each symbol encountered in the course of compilation. This would be more difficult to do with the interpreter because of possible conflicts between program symbols and compilation

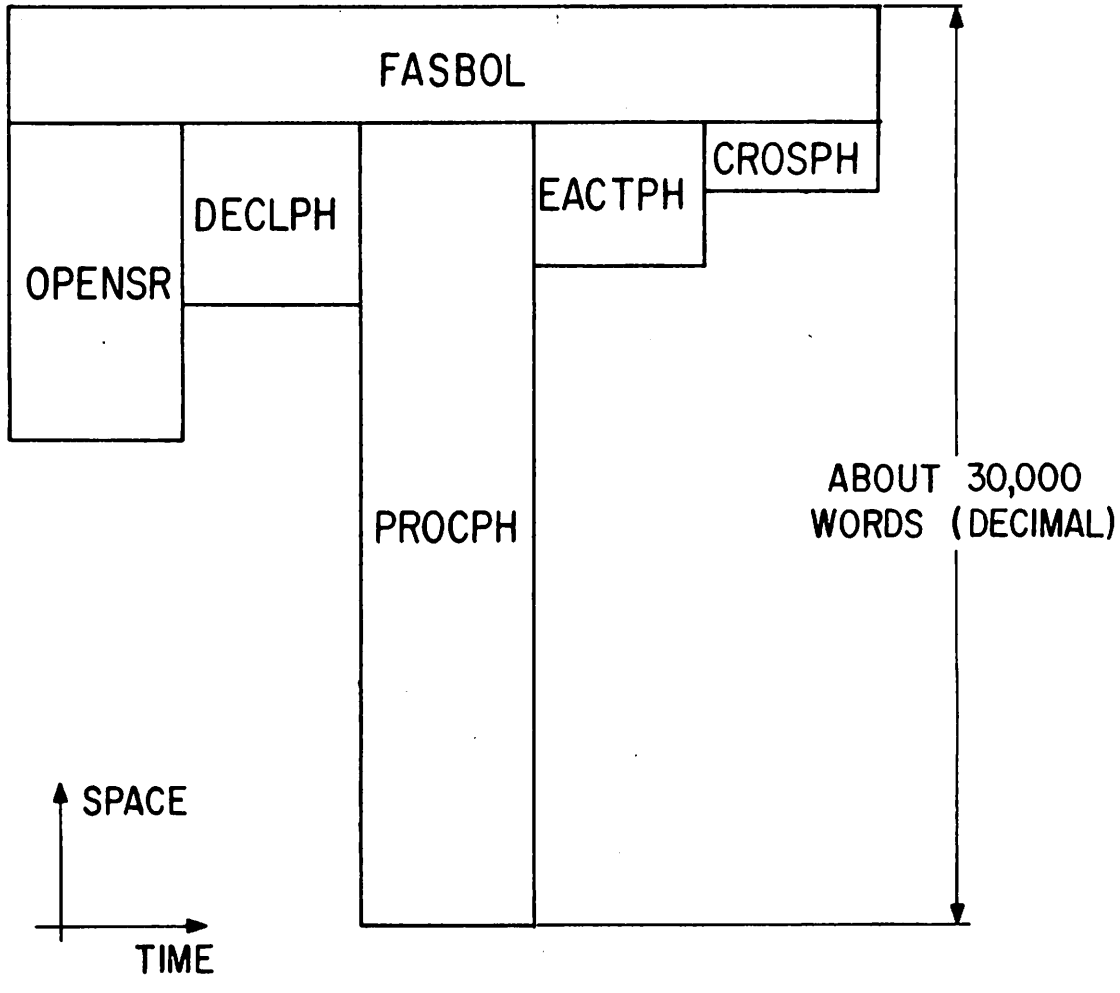


Figure 7

Space-Time Map of FASBOL Compiler

symbols, but the FASBOL compiler does not use any indirected natural variables and so the problem does not arise; indirected labels, which are used inside the compiler, are treated as separate entities by the symbol table library routine.

When a new symbol is encountered by the compiler, a "SYM" datatype (described below) is assigned to the created variable in the run-time symbol table. Symbols already entered are detected by a non-null value for their corresponding created variable. All the entries are chained together to provide access to all defined symbols at the end of compilation; in addition, symbols referenced while cross-referencing is in effect (-CROSSREF) have a cross-reference chain head ("CRS" datatype) inserted between them and the next symbol on the symbol chain. The cross reference chain consists of a series of "CRS" datatypes which contain the statement number and reference type for each reference.

The SYM datatype consists of six fields:

1. NEXT is the next SYM or CRS datatype on the chain.
2. NAME is the string corresponding to the symbol.
3. ATRB is an integer containing the attributes of the symbol usage, compressed into 36 bits.
4. INAM is a string representation of a unique number corresponding to the symbol.
5. XNAM (optional) is a string corresponding to the external name of a label variable or function for the symbol, if it has been so declared. Only one of the three can be external.

6. CHNS (optional) chain head for field or datatype chain, or TYPE*8 + number of args for primitive functions.

Any symbol may represent several different types of uses. For instance, the symbol "ABC" could be used as a variable and a literal string. Some uses are mutually exclusive; for instance no symbol can be both an integer constant and a variable (but it can be a label). With the exception of variables, labels, or functions with external names ("ENTRY" or "EXTERNAL" declarations), the (internal) name associated with a particular type of reference is generated by preceding the internal name number (INAM field) with the appropriate letter for that type. Table 3 gives the various types and the prefix and cross-reference code associated with each.

Type	Prefix	X-Ref Code
LABEL	L	0
VARIABLE	V	1
FUNCTION/FIELD	F	2
DATATYPE	D	2
STRING LITERAL	S	4
INTEGER LITERAL	I	5
REAL LITERAL	R	5
BREAK TABLE	B	7
VARIABLE NAME	N	3
NUMERICAL DESCRIPTOR	K	6

Table 3
Symbol Types

When the symbol chain is traversed in EACTPH, the attributes of each symbol determine the data code that is generated for it.

4.3 Declaration Phase

The declaration phase reads in statements until it reaches one that is not a DECLARE or OPTION pseudofunction call. For each declaration, it recognizes the type and branches to the section which handles that type. Symbols are entered, attributes set, and control passed back to the main loop. In the case of "ENTRY.FUNCTION" and "ENTRY.FORTRAN.FUNCTION", the entry code is generated. Options are treated similarly, and set flags within the compiler. Before returning to the main program, the initialization code for the compiled program is generated.

4.4 Executable Statement Phase

The executable statement phase reads in statements until it reaches an END label. Each statement is separated into a label field, a body, and a goto field. First the label, then the goto, and finally the body is processed.

4.4.1 Labels and Gotos

Labels, if any, generate a corresponding label in the output code. Goto fields are parsed and generate a goto code tree which is saved. A code tree is a simple binary tree whose end branches contain one or more lines of code. This is a more convenient and efficient way of producing code for a complex expression than either emitting the code as it is produced, or somehow concatenating it all into one string. How these code trees are built and output is explained in the next section,as well as the process by which gotos which are not simple identifiers or indirect literals are handled.

When a goto has been processed, the global fail point for the statement is known, and it is empty if there is no goto.

4.4.2 Statement Bodys

A non-empty statement body is separated into its major components in order to determine whether it is a degenerate, assignment, match, or replacement statement; assignment and replacement statements are detected by the appearance of an "Equals" sign, with the replacement having a top-level concatenation before the equals sign; Degenerate and match statements are similarly differentiated, but without an equals sign. Code trees are generated for each of the major components (VARIABLE, OBJECT, SUBJECT, PATTERN), and joined as a new tree. After the entire body has been processed, and a global fail created for statements that could fail but had no goto field, the body code tree and the goto code tree are output by calling a routine that does a bottom-up, left-right walk of each tree, emitting lines of code as they are encountered. The code trees are generated by creating "NOD" datatypes with the FRNT field and BACK field containing code or code trees which should be output in that order. For example, a code tree for A * 2 + B might be NOD(NOD("LOAD A", "MULT 2"), "ADD B") , where the "NOD" function produces an instance of the "NOD" datatype with the two arguments supplied for the FRNT and BACK fields.

Each major component of the expression (and complex gotos) is first parsed to produce a text tree [12] which contains a complete description of the operator structure, expression types

(see below), and symbol table entries. Based upon the resultant expression type, global optimization (i.e., deciding a statement is dedicated) can be done, and appropriate code-generation functions called to walk the text tree and produce the output tree.

4.4.3 Parsing

An expression is considered to have five possible types:

(a) Type 0 indicates arithmetic involving only undedicated variables with no possible input associations.

(b) Type 1 indicates integer arithmetic.

(c) Type 2 indicates real arithmetic.

(d) Type 3 indicates an expression that cannot be dedicated.

(e) Type 4 indicates an explicit pattern expr.

There are three parsing routines: One parses expressions with binary operators (including blanks for concatenation), and uses a straightforward table-driven operator precedence scheme [13]; another, called by the first, parses "Elements" (unary operations, variables, literals, parenthesized expressions, function calls, array references); and a third parses parameter lists (which involves calling the first again). The text tree is built from the bottom up, with each combination of types yielding a result type. The text tree is built with three types of nodes (datatypes) which are returned as value by the parse routines; ELN(OPTY,SBJT,PVAL) for element nodes, BON(OPTY, LFTS, RGTS) for binary operator nodes, and PLN(NXTL, PARP,PVAL) for parameter list nodes. The ELN datatype for an element holds the type (0-4) and an indicator of the nature of the element

in the OPTY field; the SBJT (subject) field holds a datatype appropriate to the nature of the element, and the PVAL field the position of the element in the statement (for any subsequent error messages). For example, a dedicated integer identifier would be of type 1 and have its SYM (symbol table entry) datatype in the SBJT field; a parenthesized expression would have the BON or ELN for the expression in the SBJT field and a type equal to that of the BON or ELN, and a programmer-defined function call would be of type 3 and have the PLN for its name and parameter list in the SBJT field.

The BON datatype for a binary operation holds the resultant type and operation in OPTY, the left side text tree in LFTS, and the right side text tree in RGTS. The binary operation parsing program builds the tree differently for operators that are left-associative (+, -, *, /, ., $) and right associative (**, SPACE, !) when it encounters more than one operator of the same precedence on the same level; the chief use being to make it easier to generate code for truly right-associative operations (like **) and for N-ary operations such as concatenation and !. The resultant type for a binary operation is extracted from a table, given its left and right types (as it is for unary operations, given the right side type).

A parameter list text tree consists of a chain of PLN datatypes, the first for the function or array variable, and then one for each parameter expression. PARP for the first holds the SYM for the function/array variable, and PVAL holds the resultant type (MAX) of the parameter types and the number of parameters; PARP

for each parameter holds the BON/ELN for that expression.

Each parse routine generates a node composed of elementary material or nodes passed to it by deeper parse routines. It advances the parse pointer past the terminal(s) that it has recognized and passes the node back as value to whoever called it.

4.4.4 Code Generation

The code generation routines operate on the same principles as suggested by Elson and Rake [12], but are not as general (or as regular). They perform a top-down context-sensitive walk over the text tree, and there are a large number (accounting for the size of PROCPH) of routines, to suit different types of expressions and/or optimizable cases. The code generated is highly machine-dependent and will not be discussed here, but some aspects emerge that are of general interest.

The simplest code-generation routine is called, given as argument a node in the text tree, and expected to return a code tree corresponding to that text. A further step is to pass information down to it, either as an argument or a global variable, which enables it to generate code based on some circumstance above it in the text tree. For example, a global variable is initially set to the statement failpoint, but set to the fail routine after ES-pushing code has been generated, and reset back to the statement failpoint after ES is considered unstacked. This global variable is used by the routine which generates code for dedicated predicates to produce a jump to either the statement failpoint or the fail routine depending upon the state of ES due to expressions above the

the predicate in the text tree. Another step still is to have code-generating functions pass back not only the code tree, but information about it. For instance, dedicated arithmetic routines pass back information concerning the arithmetic result of the code tree: location in storage or in a particular register, and whether in true or complement form. Even information not immediately usable can be passed back up the tree, such as the fact that some element might fail, or recursion could take place.

4.5 End-Action Phase

The end-action phase generates the exit code for the compiled program, and traverses the symbol list, generating any variable storage or literals that are required. A variable, for instance, generates one word labeled with the variable name (Vxxxxx), and initialized to zero (null). If the name datatype of the variable was also used in the program, a name datatype, labeled with the name name (Nxxxxx) is also generated. All items that are "INDIRECT" also generate a parameter word (under a separate location counter) used by the run-time symbol table initialization routine to enter the item (LAB,VAR,FUN) in the symbol table.

4.6 Cross-Reference Phase

This phase also traverses the symbol chain and orders the symbols lexically into a binary tree. Then a routine walks the tree in order and for each symbol, prints out the attributes of the symbol and references to each type; the statement number where a

label is defined, and where an undedicated variable is stored into, for instance, is also given.

APPENDIX I

FASBOL SYNTAX

Explanation of Syntax Notation

One problem with the usual syntax specification for SNOBOL4 (i.e., in the Bell Labs Manual, pp. 181-184), is that it is too general and permits statements which are, semantically speaking, patently illegal. The interpreter resolves these problems at execution time, but the compiler must do so at compile time. The following additions and modifications to the BNF notation are intended to assist in more closely specifying the syntax that the compiler will accept. Unfortunately, this will still be a superset of the actual set, given a specific set of declarations.

I. Only underlined symbols represent terminal elements. If the underlined symbol is nonalphabetical or uppercase, it stands for itself. Underlined lower-case alphabetic strings stand for nonprintable terminals.

II. Braces, {}, denote a grouping of syntatic units and are used instead of parentheses for the sake of clarity.

III. Square brackets, [], denote an optional appearance of the enclosed syntatic unit.

IV. Three periods, ..., denote optional repetition of the immediately preceding syntatic element (i.e., it can be repeated 0,1, ..., N times).

V. The not sign, ¬, denotes the specific ruling out* of the immediately following syntactic element.

VI. Non-terminal elements are represented by symbols consisting of lower-case alphanumerics and periods.

VII. Syntactic units and elements are best described by the following:

syntactic.element ::= TERMINAL ELEMENT |
[syntactic.element] | ¬ syntactic.element |
syntactic.element ...

syntactic.unit ::= syntactic.element | syntactic.element
spaces syntactic.unit | syntactic.element ⊥ syntactic.unit

With the following precedence (highest to lowest)

4.	¬	(unary)	negation
3.	...	(unary)	repetition
2.	space	(binary)	concatenation
1.	⊥	(binary)	alternation

*¬ is used for notational convenience, as in excluding string quotes from within their scope, or to eliminate structure leading to ambiguous parses. An example of the latter occurs with the non-terminal field, which, though in appearance identical to a function-call, is treated by the compiler in a completely different fashion. Without the ¬, there would always be at least two possible parses for an expression containing the form A(B).

Syntax for FASBOL

```
program ::= [option | declaration | comment]...[execute.body]
            end.statement
option ::= bl OPTION([bl] option.type [bl]) eos
declaration ::= bl DECLARE([bl] declaration.type [bl]) eos
comment ::= * [char]... eol | - [bl] control.type [bl] eol
execute.body ::= statement [statement]...
end.statement ::= END [bl label] eos
bl ::= blank [bl] | eol {+ | .} [bl]
eos ::= [bl] {; | eol}
statement ::= comment | [label.field] [statement.body] [goto.field]
              eos
label.field ::= ¬ {END bl} label
goto.field ::= bl : [bl] {goto | S goto [bl] [F goto] |
               F goto [bl] [S goto]}
goto ::= ([bl] {identifier | $ string.primary} [bl])
statement.body ::= degenerate | assignment | match | replacement
degenerate ::= bl string.primary
assignment ::= bl variable equals bl {string.expression |
               ¬ {string.primary [bl string.primary]...}
               pattern.expression}
match ::= bl string.primary bl pattern.expression
replacement ::= bl variable bl pattern.expression equals
                [bl string.expression]
equals ::= bl =
variable ::= identifier | string.variable | procedure.call.form
```

```
pattern.expression ::= conjunction [bl ! bl conjunction]...

conjunction ::= pattern.term [bl pattern.term]...

pattern.term ::= pattern.primary [{bl . bl | bl $ bl}
                 pattern.variable]...

pattern.primary ::= pattern.primitive | @ pattern.variable |
                    [*] string.primary | ( [bl] pattern.expression
                    [bl])

pattern.variable ::= [*] variable

string.expression ::= sum [bl sum] ...

sum ::= term [{bl + bl | bl - bl} term] ...

term ::= factor [{bl * bl | bl / bl} factor]...

factor ::= string.primary [bl ** bl string.primary]...

string.primary ::= ¬ pattern.primitive {literal | identifier |
                   string.variable | procedure.call | {? | \ |
                   - | +} string.primary | . variable | (
                   [bl] string.expression[bl])}

literal ::= integer.literal | real.literal | string.literal

string.variable ::= $ string.primary | & keyword | array.element |
                    field

procedure.call ::= expression.primitive | ¬ {expression.primitive |
                   field | ITEM ([bl] parameter.list [bl])}
                   procedure.call.form

procedure.call.form ::= {¬ APPLY identifier | APPLY}
                        ([bl][parameter.list] [bl])

parameter.list ::= string.expression [pc string.expression]...

pc ::= [bl] , [bl]
```

```
array.element ::= identifier < [bl] [parameter.list] [bl] > |
                  ITEM ([bl] parameter.list [bl])
field ::= ¬{procedure.call | ITEM ([bl] parameter.list [bl])}
          identifier known field ([bl]
          string.expression [bl])
identifier ::= letter [letter | digit | . | - ]...
label ::= ¬{blank | ; | - | + | . | *} char [¬{blank | ;}
          char]...
integer.literal ::= digit [digit]...
real.literal ::= digit [digit]... . [digit]...
string.literal ::= ' [¬' cont.char]... ' | # [¬# cont.char]... #
cont.char ::= char | eol {+ | .}
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 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
eol ::= end-of-line
char ::= any character

option.type[1] ::= 'QUICKSTORE' | 'NO.LOOK-AHEAD' | 'NO.STNO' |
                   'TIMER' | 'HASHSIZE=' integer.literal '
declaration.type[1] ::=
  'INDIRECT.VARIABLE' pc {ALL | ' identifier.list '}
| 'NOT.INDIRECT.VARIABLE' pc ' identifier.list '
| 'INDIRECT.FUNCTION' pc {ALL | ' identifier.list '}
| 'NOT.INDIRECT.FUNCTION' pc ' identifier.list '
| 'INDIRECT.LABEL' pc {ALL | ' label.list '}
| 'NOT.INDIRECT.LABEL' pc ' label.list '
```

| 'INPUT.ASSOCIATIONS' pc ' identifier.list '

| 'OUTPUT.ASSOCIATIONS' pc ' identifier.list '

| 'FIELD' pc ' identifier.list '

| 'STRING' pc ' string.specifier.list '

| 'INTEGER' pc ' identifier.list '

| 'REAL' pc ' identifier.list '

| 'PATTERN' pc ' identifier.list '

| 'SNOBOL.SUBPROGRAM' pc ' restricted.identifier '

| 'EXTERNAL.VARIABLE' pc ' restricted.identifier.list '

| 'EXTERNAL.FUNCTION' pc ' restricted.identifier.list'

| 'EXTERNAL.LABEL' pc ' restricted.label.list '

| 'ENTRY.VARIABLE' pc ' restricted.identifier.list '

| 'ENTRY.FUNCTION' pc ' entry.define.prototype ' [pc ' label^{2};'

| 'ENTRY.LABEL' pc ' restricted.label.list '

| 'EXTERNAL.FORTRAN.FUNCTION' pc ' FORTRAN.identifier.list '

[pc 'OPEN']

| 'ENTRY.FORTRAN.FUNCTION' pc ' FORTRAN.prototype ' pc ' label^{2};'

[pc ' label^{2};']

control.type ::=

LIST | UNLIST | NOCODE | CODE | EJECT | FAIL | NOFAIL

| NOCROSS | CROSSREF | NOASM | LISTASM | SPACE [bl integer.literal]

| NEWSTNO bl ⌐{0[0]...} integer.literal

```
identifier.list ::= identifier [pc identifier]...

label.list ::= label[2] [bl label[2]]... ;

string.specifier.list ::= string.specifier [pc string.specifier]...

string.specifier ::= identifier ([bl] integer.literal [bl])

restricted.identifier.list ::= restricted.identifier
                               [ pc restricted.identifier]...

restricted.label.list ::= restricted.label [pc restricted.label]...

entry.define.prototype ::= restricted.identifier ([bl]
                           [identifier.list] [bl])
                           [[bl] identifier.list]

FORTRAN.identifier.list ::= FORTRAN.identifier [pc FORTRAN.identifier]...

FORTRAN.prototype ::= FORTRAN.identifier ([bl]
                      [dedicated.identifier.list] [bl])

dedicated.identifier.list ::= dedicated.identifier
                              [pc dedicated.identifier]...

FORTRAN.identifier ::= restricted.identifier[= {INTEGER | REAL}]

dedicated.identifier ::= identifier [= {INTEGER | REAL}] |
                         string.specifier

restricted.identifier ::= letter [ln [ln [ln [ln [ln]]]]]

restricted.label ::= letter [lnd [lnd [lnd [lnd [lnd]]]]]

ln ::= letter | digit

lnd ::= letter | digit | $
```

```
pattern.primitive ::= FAIL | FENCE | ABORT | ARB | BAL | SUCCEED
                    | REM | {LEN | TAB | RTAB | POS | RPOS
                    | SPAN | NSPAN | BREAK | BREAKX | ANY
                    | NOTANY } ([bl]{string.expression
                    | * string.primary} [bl])
                    | STRVAL ([bl] {string.expression
                    | * string.primary} [bl])
                    | ARBNO ([bl] pattern.expression [bl])

keyword ::= STFCOUNT | LASTNO | STNO | FNCLEVEL | STCOUNT |
            RTNTYPE | ALPHABET | ABEND | ANCHOR | FULLSCAN |
            MAXLNGTH | STLIMIT

expression.primitive ::= {OPSYN | DETACH | INPUT | OUTPUT |
                         REPLACE} ([bl] [parameter.list][bl]) |
                         DEFINE ([bl] ' define.prototype '
                         [pc string.expression] [bl]) | DATA
                         ([bl] ' data.prototype ' [bl])
                         |{DATE | TIME | EJECT | INTEGER | SIZE
                         | TRIM | DATATYPE | COPY | ENDFILE
                         | REWIND | BACKSPACE | PROTOTYPE
                         | COLLECT | EXTIME | BUFSIZ | REVERS
                         | REMCOR | DIFFER | IDENT | LGT
                         | CONVERT | ARRAY | DUPL | LPAD | RPAD
                         | SUBSTR | INSERT} ( [bl][parameter.list]
                         [bl])
                         |{NOT | LT | LE | EQ | NE | GE | GT | AND
                         | OR | XOR | RSHIFT | LSHIFT} ([bl]
                         [parameter.list] [bl])
```

define.prototype ::= identifier ([bl] [identifier.list] [bl])
[bl] [identifier.list]

data.prototype ::= identifier ([bl] identifier.list [bl])

NOTES: 1. Any pair of single quote (') brackets can be replaced by a pair of double quote (#) brackets.

2. label is additionally restricted to not contain a quote of the type that form the string brackets.

Run-Time Syntax

array.prototype ::= dimension [, dimension]...

dimension ::= signed.integer [: signed.integer]

signed.integer ::= [+ | -] integer.literal

APPENDIX 2

INTERNAL AND I/O ERROR MESSAGES

Internal

1. Increment use count of unused block.
2. Release of unused block
3. Underflow of RS
4. Underflow of ES
5. Missing field definition for datatype.
6. Underflow of SS
7. Underflow of AS
8. Overflow of CV
9. Underflow of CV
10. Imaginary conditional or immediate assignment.
11. Overflow in concatenation, too many elements.

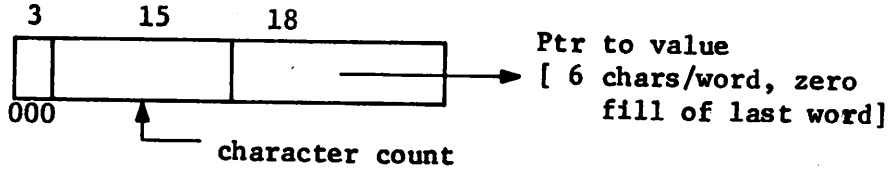
I/O

0. Association length too long or negative.
1. Negative unit No.
2. Unit No. too large.
3. Association table overflow.
4. Illegal unit (device)
5. Unassigned unit.
6. Illegal operation on device.
7. Unrecoverable tape read error.
8. Unrecoverable drum read error.
9. Drum movement exceeded file limits.

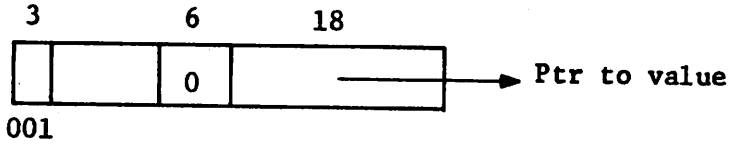
APPENDIX 3

DESCRIPTOR FORMATS

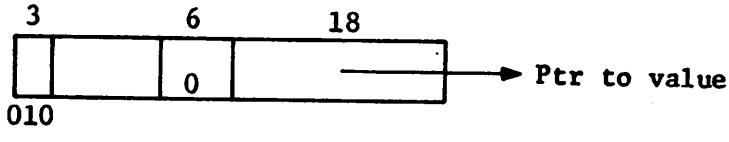
STRING



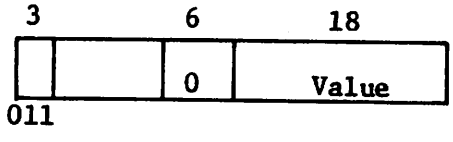
INTEGER



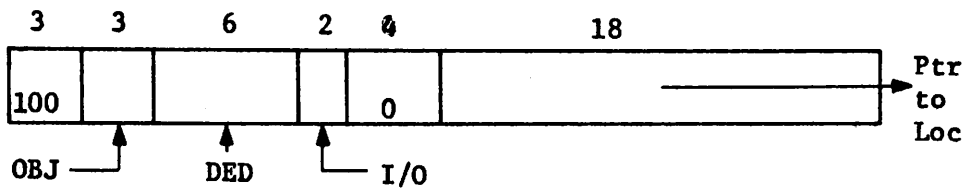
REAL



IMMEDIATE INTEGER



NAME

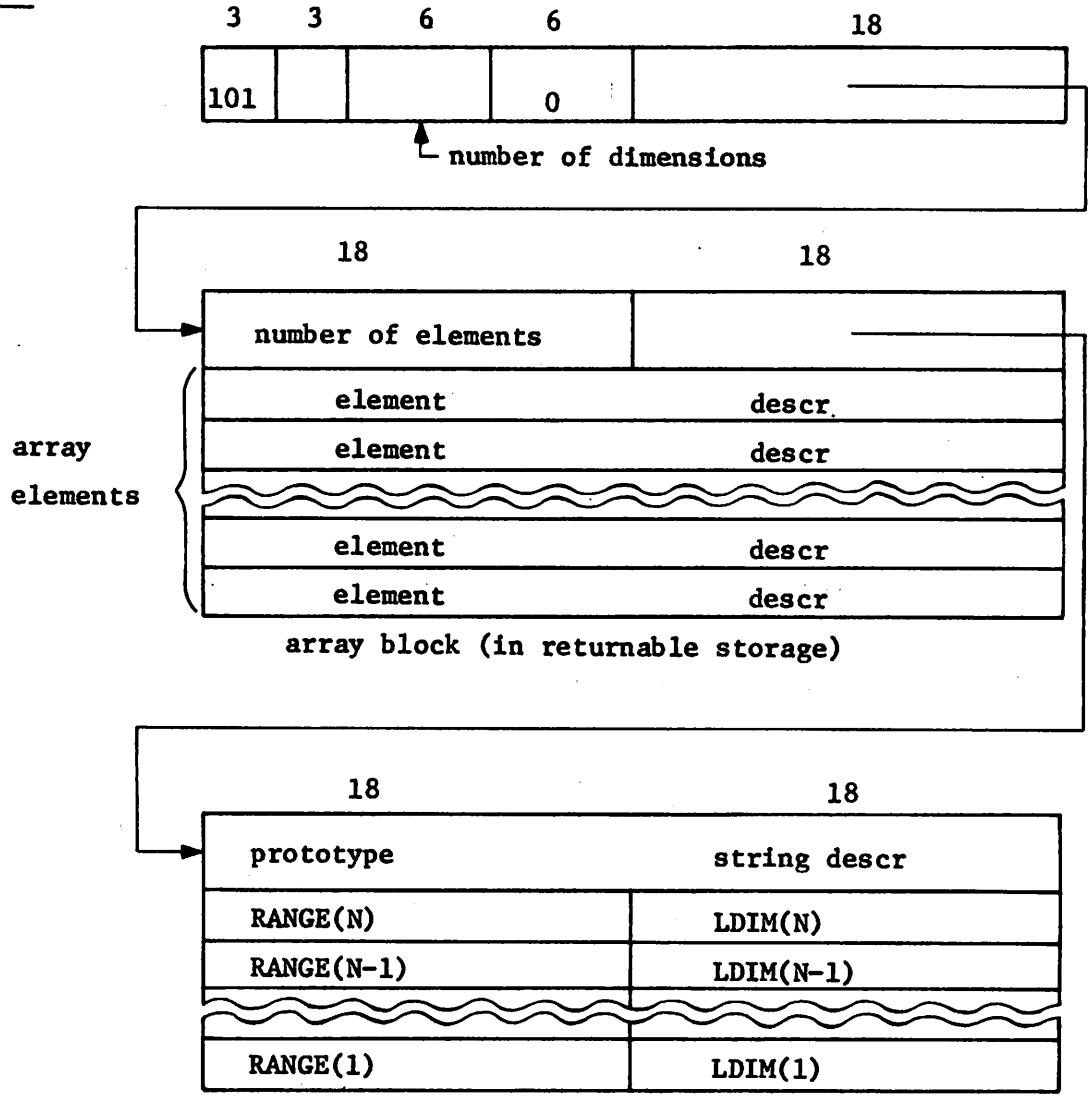


OBJ:	0	Variable		
	1	Label	(DED = 0)	(I/O = 0
	2	Function	(DED = 0)	(I/O = 0)

DED:	0	Not Dedicated	
	1	Dedicated String	(I/O = 0)
	2	Dedicated Integer	(I/O = 0)
	3	Dedicated Real	(I/O = 0)
	4	Keyword	(I/O = 0)
	5	Dedicated Pattern	

I/O:	0	No I/O	Association	Possible
	1	Input	"	"
	2	Output	"	"
	3	Both	Associations	"

ARRAY



RANGE(I) = Size of Dimension I

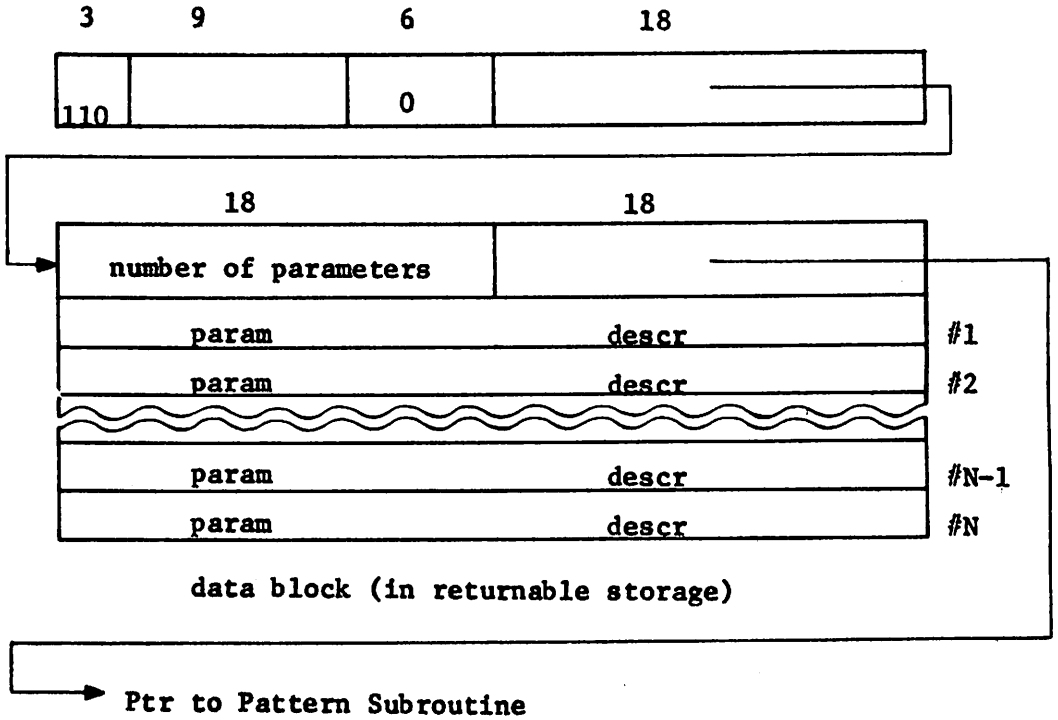
LDIM(I) = Lower Bound of Dimension I

Mapping Function

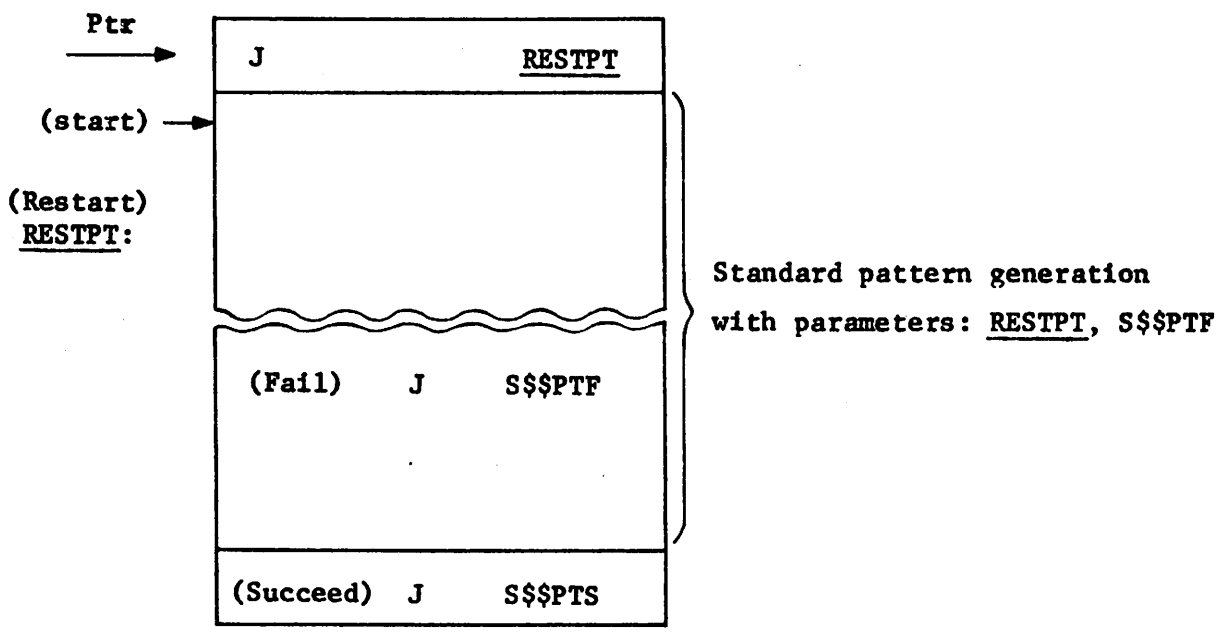
LOC(ARRAY < I1, I2, ..., IN >) =

$$\left[\begin{array}{l} \text{LOC (PTR(ARRAY))} + 1 + \text{I1} - \text{LDIM[1]} \\ \quad + \text{RANGE(1)} * (\text{I2} - \text{LDIM(2)} \\ \qquad \ldots \\ \qquad + \text{RANGE(N-1)}*(\text{IN-LDIM(N)} \\ \qquad) \ldots) \end{array}\right]$$

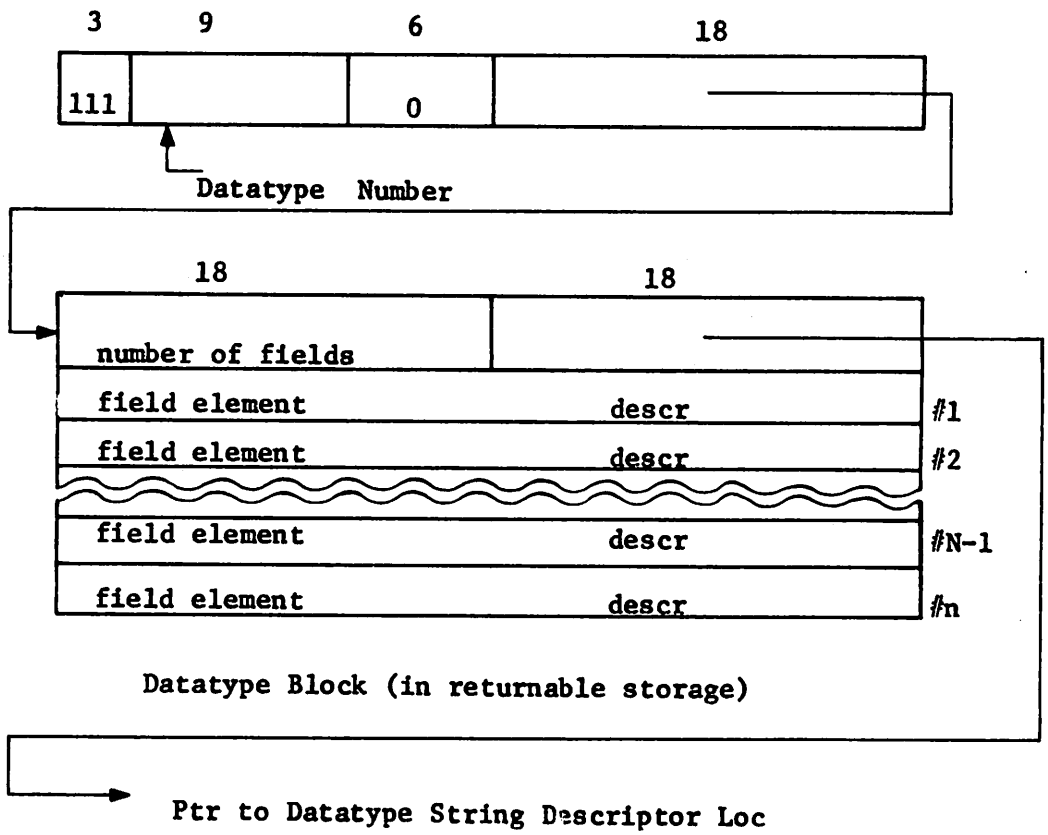
PATTERN



PATTERN SUBROUTINE: (may be in returnable storage if created dynamically by a concatenation)



PROGRAMMER-DEFINED DATATYPE



APPENDIX 4

```
GO       SNO,*      RECOG                                          10 NOV 71    00:53:27.141
      ***********************************
      ** FASBOL COMPILER VERS 1.0 (JUNE,1971) **
      ***********************************

      THIS COMPILATION WAS DONE ON 10 NOV 71 AT 00:53:27
```

```
000001             *
000002             *       THIS PROGRAM IS A SYNTACTIC RECOGNIZER FOR SNOBOL4 STATEMENTS.
000003             *
000004             *       FIRST A SERIES OF PATTERNS IS BUILT CULMINATING IN A PATTERN
000005             *       WHICH MATCHES ONLY SYNTACTICALLY CORRECT STATEMENTS. CARD IMAGE
000006             *       S ARE THEN READ IN AND PROCESSED. INCORRECT STATEMENTS ARE
000007             *       IDENTIFIED BY AN ERROR MESSAGE
000008             *
000009             *       THE FUNCTION OPT FORMS A PATTERN THAT MATCHES EITHER NULL OR IT
000010             *       S ARGUMENT
000011             *
000012          1          DEFINE('OPT(PATTERN)')
000013             *
000014          2          LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
000015             *
000016          3          DIGITS = '0123456789'
000017          4          ALPHANUMERICS = LETTERS DIGITS
000018          5          BLANKS = SPAN(' ')
000019          6          INTEGER = SPAN(DIGITS)
000020          7          REAL = SPAN(DIGITS) '.' NSPAN(DIGITS)
000021          8          IDENTIFIER = ANY(LETTERS) NSPAN(ALPHANUMERICS '-.')
000022          9          UNARY = ANY('+-&.$*?\%')
000023         10          BINARY = ANY('-+.$*!/*') ! '**'
000024         11          BINARYOP = BLANKS OPT(BINARY BLANKS)
000025         12          ULGALPHABET = &ALPHABET
000026         13          UNOALPHABET '#' =
000027         14          UNOALPHABET *'B =
000028         15          DLITERAL = '*' SPAN(UNOALPHABET #'#) '#'
000029         16          SLITERAL = #'# SPAN(UNOALPHABET '#') #'#
000030         17          LITERAL = SLITERAL ! DLITERAL ! INTEGER ! REAL
000031         18          ELEMENT = OPT(UNARY) (IDENTIFIER ! LITERAL !.*FUNCTION-CALL
000032         18 .        ! '(' (*EXPRESSION ! NSPAN(' ')) ')' ! *ARRAY-REF)
```

```
000033         19          OPERATION = *ELEMENT BINARYOP (*ELEMENT ! *EXPRESSION)
000034         20          EXPRESSION = NSPAN(' ') (*ELEMENT ! *OPERATION ! NULL)
000035         20 .        NSPAN(' ')
000036         21          ARG-LIST = ',' *ARG-LIST
000037         22          ARG-LIST = *EXPRESSION OPT(ARG-LIST)
000038         23          FUNCTION-CALL = IDENTIFIER '(' *ARG-LIST ')'
000039         24          ARRAY-REF = IDENTIFIER '<' *ARG-LIST '>'
000040         25          LABEL = ANY(ALPHANUMERICS) (BREAK(' :') ! REM)
000041         26          LABEL-FIELD = OPT(LABEL)
000042         27          GOTO = '(' EXPRESSION ')' ! '<' EXPRESSION '>'
000043         28          GOTO-FIELD =  BLANKS ':' FENCE NSPAN(' ') (GOTO ! 'S'
000044         28 .        GOTO ! 'F' GOTO ! 'S' GOTO NSPAN(' ') 'F' GOTO
000045         28 .        ! 'F' GOTO NSPAN(' ') 'S' GOTO) NSPAN(' ')
000046         29          GOTO-FIELD = OPT(GOTO-FIELD)
000047         30          RULE = BLANKS ELEMENT (BLANKS '=' OPT(BLANKS EXPRESSION)
000048         30 .        ! OPT(BLANKS EXPRESSION OPT(BLANKS '=' OPT(BLANKS EXPRESSION
000049         30 .        ))))
000050         31          RULE       = OPT(RULE)
000051         32          EOS        = RPOS(0) ! ';'
000052         33          STATEMENT = LABEL-FIELD RULE GOTO-FIELD EOS
000053                  *
000054                  *   THE PATTERN FOR RECOGNIZING STATEMENTS IS NOW FORMED. THE
000055                  *   PROGRAM TO ANALYZE INPUT CARDS FOLLOWS.
000056                  *
000057         34          COMMENT = ANY('*-')
000058         35          CONTINUE = ANY('.+') . CC
000059         36          INPUT('INPUT',5,72)
000060         37          &ANCHOR = 1
000061         38          EOF =
000062                  *
000063                  *   INITIALIZE PROCESS FROM FIRST CARD
000064                  *
000065         39 READI    IMAGE = TRIM(INPUT)                                  :F(END)
000066         40          OUTPUT = '      ' IMAGE
000067                  *
000068                  *   DO NOT PROCESS COMMENT OR CONTINUE CARDS.
000069                  *
000070         41          IMAGE COMMENT                                        :F(READC)
000071         41 .                                                             S(READI)
000072         42 NEXTST   IDENT(EOF)                                           :F(END)
000073         43          OUTPUT    = '      ' LINE
000074         44          IMAGE = LINE
000075         45 READC    LINE = TRIM(INPUT)                                   :F(ENDGAME)
000076         46          LINE COMMENT                                         :S(PRINT)
```

```
000077        47           LINE CONTINUE =                                  :F(ANALYZE)
000078        48           OUTPUT = '    ' CC LINE
000079        49           IMAGE = IMAGE LINE                               :(READC)
000080        50 ANALYZE   IMAGE STATEMENT =                                :F(ERROR)
000081        51           DIFFER(IMAGE)                                    :S(ANALYZE)
000082        52           OUTPUT = '<<< NO SYNTACTIC ERROR>>>'
000083        53 SKIP      OUTPUT =                                         :(NEXTST)
000084                   *
000085                   *       IF AN ERRONEOUS STATEMENT IS ENCOUNTERED IN A STRING OF
000086                   *       STATEMENTS SEPARATED BY SEMICOLONS, SUBSEQUENT STATEMENTS ARE
000087                   *       NOT PROCESSED
000088                   *
000089        54 ERROR     OUTPUT = '<<< SYNTACTIC ERROR >>>'               :(SKIP)
000090                   *
000091        55 PRINT     OUTPUT    = '     ' LINE                         :(READC)
000092        56 ENDGAVE   EOF = 1                                          :(ANALYZE)
000093                   *
000094                   *
000095        57 OPT       OPT = NULL ! PATTERN                             :(RETURN)
000096        58 END
```

TOTAL COMPILATION TIME: 2.42 MS., 0 ERROR DIAGNOSTICS

```
*
*       A VARIETY OF CORRECT AND INCORRECT SNOBOL4 STATEMENTS FOLLOW
-LIST
    COMPUTE  X    = Y + 3 ** -'2'
<<< NO SYNTACTIC ERROR>>>

             X  =   Y+Z
<<< SYNTACTIC ERROR >>>

             ELEMENT<I,J>= ELEMENT<I,-J> + ELEMENT<-I,J>
<<< SYNTACTIC ERROR >>>

             A<X,Y,Z + 1>  =  F(X,STRUCTURE-BUILD(TYPE,LENGTH + 1))
<<< NO SYNTACTIC ERROR>>>

    SETUP   PAT1.  = (BREAK(',:') $ FIRST ! SPAN(' .') $ SECOND
                   . VALUE ARGNO(BAL ! LEN(1))   :($SWITCH)
<<< NO SYNTACTIC ERROR>>>

             DEFINE('F(X,Y)')
<<< SYNTACTIC ERROR >>>

             L  =  LT(N,P<J> L + 1
<<< SYNTACTIC ERROR >>>

    NERUNE-TRIAL  X   =   \COORD<1,K>  X * X
<<< NO SYNTACTIC ERROR>>>

             TRIM(INPUT)   PAT1   :S(OK)   :F(PAD)
<<< SYNTACTIC ERROR >>>

        X   =   3.01; Y = 2.    ; Z  =  X * -Y
<<< NO SYNTACTIC ERROR>>>


NORMAL TERMINATION AT LEVEL    0
LAST STATEMENT EXECUTED WAS    42 - MAIN

SNOPOL STATISTICS SUMMARY-

          524 MS. EXECUTION TIME
          161 STATEMENTS EXECUTED,      32 FAILED
```

```
JJ       S:0,4      SORT                                                    10 NOV 71    00:53:42.164
      ****************************************
      ** FASBOL COMPILER VERS 1.0 (JUNE,1971) **
      ****************************************

      THIS COMPILATION WAS DONE ON 10 NOV 71 AT 00:53:42
```

```
000001             *
000002             *                    TOPOLOGICAL SORT
000003             *
000004             *  MAPS A PARTIAL ORDERING OF OBJECTS INTO A LINEAR ORDERING
000005             *
000006             *       A(1), A(2), ..., A(N)
000007             *
000008             *  SUCH THAT IF A(S) < A(T) IN THE PARTIAL ORDERING, THEN S < T.  (CF.
000009             *  D.E. KNUTH, THE ART OF COMPUTER PROGRAMMING, VOLUME 1, ADDISON-WESLEY,
000010             *  MASS. 1968, P. 262)
000011             *
000012          1          DATA('ITM(COUNT,TOP)')
000013          2          DATA('NOD(SUC,NEXT)')
000014          3          DEFINE('DECR(X)')
000015          4          DEFINE('INDEX(TAU)')
000016             *
000017             *  READ IN THE NUMBER OF ITEMS, N, AND GENERATE AN ARRAY OF ITEMS.
000018             *
000019             *  EACH ITEM HAS TWO FIELDS, (COUNT,TOP), WHERE
000020             *       COUNT = NO. OF ELEMENTS PRECEDING IT,
000021             *       TOP = TOP OF LIST OF ITEMS SUCCEEDING IT.
000022             *
000023          5          N          = TRIM(INPUT)
000024          6          X          = ARRAY('0:' N)
000025             *
000026             *  INITIALIZE THE ITEMS TO (0,NULL).
000027             *
000028          7  T1       X<I>       = ITM(0,'')                          :F(T1A)
000029          8          I          = I + 1                              :(T1)
000030             *
000031             *  READ IN RELATIONS
000032             *
```

```
000033          9 T1A       OUTPUT     = ' THE RELATIONS ARE:'
000034         10 T2A       REL        = TRIM(INPUT) ','                          :F(T3A)
000035         11           OUTPUT     = REL
000036         12 T2        REL        BREAK('<') $ MU LEN(1) BREAK(',') $ NU LEN(1) =
000037         12 .                                                               :F(T2A)
000038         13           J          = INDEX(MU)
000039         14           K          = INDEX(NU)
000040                *
000041                *  SINCE MU < NU, INCREASE THE COUNT OF THE KTH ITEM AND ADD A NODE TO
000042                *  THE LIST OF SUCCESSORS OF THE JTH ITEM.
000043                *
000044         15 T3        COUNT(X<K>) = COUNT(X<K>) + 1
000045         16           TOP(X<J>) = NOD(K,TOP(X<J>))                          :(T2)
000046                *
000047                *  A QUEUE IS MAINTAINED OF THOSE ITEMS WITH ZERO COUNT FIELD. THE LINKS
000048                *  FOR THE QUEUE ARE KEPT IN THE COUNT FIELD. THE VARIABLES F,R POINT
000049                *  TO THE FRONT AND REAR OF THE QUEUE.
000050                *
000051         17 T3A
000052                *
000053                *  INITIALIZE THE QUEUE FOR OUTPUT.
000054                *
000055         18           R          = 0
000056         19           COUNT(X<0>) = 0
000057         20           Q          = 0
000058         21 T4        Q          = ?X<Q + 1> Q + 1                          :F(T4A)
000059         22           COUNT(X<R>) = EQ(COUNT(X<Q>),0) Q                     :F(T4)
000060         23           R          = Q                                        :(T4)
000061         24 T4A       F          = COUNT(X<0>)
000062                *
000063                *  OUTPUT THE FRONT OF THE QUEUE.
000064                *
000065         25           OUTPUT = ; OUTPUT = ' THE LINEAR ORDERING IS:'
000066         27 T5        OUTPUT     = NE(F,0) $F                               :F(T8)
000067         28           N          = N - 1
000068         29           P          = TOP(X<F>)
000069                *
000070                *  ERASE RELATIONS
000071                *
000072         30 T6        IDENT(P)                                              :S(T7)
000073         31           DECR(.COUNT(X<SUC(P)>))                               :S(T6A)
000074                *
000075                *  IF COUNT IS ZERO ADD ITEM TO QUEUE.
000076                *
```

```
000077        32            COUNT(X<R>) = SUC(P)
000078        33            R           = SUC(P)
000079        34 T6A        P           = NEXT(P)                          :(T6)
000080              *
000081              *  REMOVE FROM QUEUE.
000082              *
000083        35 T7         F           = COUNT(X<F>)                      :(T5)
000084              *
000085              *  FUNCTION DEFINITIONS.
000086              *
000087        36 DECR       $X          = GT($X,1) $X - 1                  :S(RETURN)
000088        37            $X          = 0                                :(FRETURN)
000089              *
000090        38 INDEX      INDEX       = .$TAU
000091        39            INDEX       = DIFFER($INDEX) $INDEX            :S(RETURN)
000092        40            TERMCT      = LT(TERMCT,N) TERMCT + 1          :F(FRETURN)
000093        41            $TERMCT     = TAU
000094        42            $INDEX      = TERMCT
000095        43            INDEX       = TERMCT                           :(RETURN)
000096              *
000097        44 T8         OUTPUT      = NE(N) ' THE ORDERING CONTAINS A LOOP.'
000098        45 END

*TOTAL COMPILATION TIME: 1405 MS., 0 ERROR DIAGNOSTICS*
```

```
 THE RELATIONS ARE:
LETTERS<ALPHANUM,NUMBERS<ALPHANUM,
BLANKS<OPTBLANKS,
NUMBERS<REAL,
NUMBERS<INTEGER,
LETTERS<VARIABLE,ALPHANUM<VARIABLE,
BINARY<BINARYOP,BLANKS<BINARYOP,
UNSALPHABET<DLITERAL,
UNSALPHABET<SLITERAL,
SLITERAL<LITERAL,DLITERAL<LITERAL,INTEGER<LITERAL,REAL<LITERAL,

 THE LINEAR ORDERING IS:
LETTERS
NUMBERS
BLANKS
BINARY
UNSALPHABET
INTEGER
REAL
ALPHANUM
OPTBLANKS
BINARYOP
SLITERAL
DLITERAL
VARIABLE
LITERAL

NORMAL TERMINATION AT LEVEL    0
LAST STATEMENT EXECUTED WAS    44 - MAIN


SNOBOL STATISTICS SUMMARY-

          75 MS. EXECUTION TIME
         456 STATEMENTS EXECUTED,      68 FAILED
```

```
EJ      SNO,*     FACTOR                                          10 NOV 71   00:53:50.795
      ***************************************
      ** FASBOL COMPILER VERS 1.0 (JUNE,1971) **
      ***************************************

      THIS COMPILATION WAS DONE ON 10 NOV 71 AT 00:53:51



000001            - CROSSREF
000002            * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
000003            *                                                                       *
000004            *        THIS PROGRAM COMPUTES AND PRINTS A TABLE OF N FACTORIAL        *
000005            *        FOR VALUES OF N FROM 1 THROUGH AN UPPER LIMIT 'NX'.            *
000006            *                                                                       *
000007            *        IT DEMONSTRATES A METHOD OF MANIPULATING NUMBERS WHICH ARE     *
000008            *        TOO LARGE FOR THE COMPUTER, AS STRINGS OF CHARACTERS.  THE     *
000009            *        COMMAS IN THE PRINTED VALUES ARE OPTIONAL, ADDED FOR READING   *
000010            *        EASE.                                                          *
000011            *                                                                       *
000012            * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
000013            *
000014            *        INITIALIZATION.
000015            *
000016          1           OPTION('TIMER')
000017          2           OPTION('QUICKSTORE')
000018          3           DECLARE('INTEGER','NX,N,NSET,I')
000019          4           DECLARE('EXTERNAL,FORTRAN,FUNCTION','MAKNXT,NUM')
000020          5           NX = 45
000021            *
000022          6           N = 1
000023          7           NSET = 1
000024            *
000025          8           OUTPUT = '          TABLE OF FACTORIALS FOR 1 THROUGH ' NX
000026          9           OUTPUT =
000027            *
000028            *         COMPUTE THE NEXT VALUE FROM THE PREVIOUS ONE.
000029            *
000030         10 L1        NUMBER = MAKNXT(N,NSET)
000031            *
000032            *         FORM A STRING REPRESENTING THE FACTORIAL.
```

```
000033              *
000034          11          I = GT(NSET,1) NSET - 1                                 :F(L7)
000035          12 L6       NUMBER = NUMBER ',' LPAD(NUM(I),3,'0')
000036          13          I = GT(I,1) I - 1                                       :S(L6)
000037              *
000038              *       OUTPUT A LINE OF THE TABLE
000039              *
000040              -CODE
000041          14 L7       OUTPUT = N '!=' NUMBER
L11011.
 LMJ 6,SSSIPN.
 +   8,14.
 L   12,=L1004.
 S   12,SSS/SP.
 L   12,V11001.
 LMJ 6,SSSCID.
 LMJ 10,SS=LES.
 L   12,S111016.
 LMJ 1.,SS=LES.
 L   12,V11010.
 LMJ 6,SSSCIC.
 +   3*/n*J.
 LMJ 6,S=xASG.
000042          15          N = LT(N,NX) N + 1                                      :S(L1)F(END)
 LMJ 6,SSSIPN.
 +   8,1=.
 L   12,V11001.
 A=  12,V11000.
 JP  12,L11017.
 L   12,V11001.
 A   12,1*0,14.
 S   12,V11001.
 J   L11003.
000043              -NOCODE
000044              *
000045              *       ERROR TERMINATION.
000046              *
000047          16 ERR      OUTPUT = N '! CANNOT BE COMPUTED BECAUSE OF TABLE OVERFLOW.'
000048          17          OUTPUT = '   INCREASE THE SIZE OF ARRAY #NUM#.'
000049              *
000050          18 END
```

```
      ,*****************************************
      ******* CROSS-REFERENCE DICTIONARY *******
      *****************************************


[SYMBOL]  (INTERNAL)
      ATTRIBUTES, STATEMENT NUMBERS


[          TABLE OF FACTORIALS FOR ] THROUGH ]  (11008)
      STRING LITERAL REF,8

[   INCREASE THE SIZE OF ARRAY #NUM#.]  (11020)
      STRING LITERAL REF,17

[END]  (11017)
      LABEL (DEFINED 18) REF,15

[ERR]  (11018)
      LABEL (DEFINED 16) REF

[GI]  (10052)
      ENTRY OR EXTERNAL FUNCTION (PRIMITIVE) REF,11,13

[I]  (11003)
      INTEGER VARIABLE REF,3,11,12,13,13,13

[LPAD]  (10033)
      ENTRY OR EXTERNAL FUNCTION (PRIMITIVE) REF,12

[LT]  (10047)
      ENTRY OR EXTERNAL FUNCTION (PRIMITIVE) REF,15

[L1]  (11009)
      LABEL (DEFINED 10) REF,15

[L6]  (11012)
      LABEL (DEFINED 12) REF,13

[L7]  (11011)
      LABEL (DEFINED 14) REF,11
```

```
[MAKEXT]  (11004)
      INTEGER FORTRAN FUNCTION REF,4,10

[N]  (11001)
      INTEGER VARIABLE REF,3,6,10,14,15,15,15,16

[NSET]  (11002)
      INTEGER VARIABLE REF,3,7,10,11,11

[NUM]  (11005)
      INTEGER FORTRAN FUNCTION REF,4,12

[NUMBER]  (11010)
      VARIABLE REF,10,12,12,14
      VARIABLE NAME REF,10,12

[NX]  (11006)
      INTEGER VARIABLE REF,3,5,8,15

[OUTPUT]  (10004)
      VARIABLE (OUTPUT ASSOC.) REF,8,9,14,16,17
      VARIABLE NAME REF,8,9,14,16,17

[! CANNOT BE COMPUTED BECAUSE OF TABLE OVERFLOW.]  (11019)
      STRING LITERAL REF,16

[!=]  (11016)
      STRING LITERAL REF,14

[,]  (11013)
      STRING LITERAL REF,12

[0]  (11015)
      STRING LITERAL REF,12

[1]  (11007)
      INTEGER CONSTANT REF,6,7,11,11,13,13,15

[3]  (11014)
      INTEGER CONSTANT REF,12
      INTEGER LITERAL REF,12

[45]  (11008)
```

```
INTEGER CONSTANT REF,5


*TOTAL COMPILATION TIME: 778 MS., 0 ERROR DIAGNOSTICS*
```

```
.II      FOR        MAKNXT,MAKNXT                                    10 NOV 71    00:53:53.827
UNIVAC 1108 FORTRAN V LEVEL   2206 0018 F5018S
THIS COMPILATION WAS DONE ON 10 NOV 71 AT 00:53:53

   FUNCTION MAKNXT      ENTRY POINT 000074

            NUM         ENTRY POINT 000107

   STORAGE USED (BLOCK, NAME, LENGTH)

         0001    *CODE    000114
         0000    *DATA    001765
         0002    *BLANK   000000


   EXTERNAL REFERENCES (BLOCK, NAME)

         0003    NERR3$


   STORAGE ASSIGNMENT FOR VARIABLES (BLOCK, TYPE, RELATIVE LOCATION, NAME)

    0001   000005 107G       0001   000013 114G       0001   000032 20L        0000 I 000000 MAKNXT     0000 I 000001 NUMBER
    0000 I 001751 NUMX



00101     1*              FUNCTION MAKNXT(N,NSET)
00103     2*              DIMENSION NUMBER(1000)
00104     3*              DATA NUMBER(1)/1/
00106     4*              DO 10 I=1,NSET
00111     5*     10       NUMBER(I) = NUMBER(I)*N
00113     6*              DO 20 I=1,NSET
00114     7*              IF (NUMBER(I).LT.1000) GO TO 20
00120     8*              NUMX = NUMBER(I)/1000
00121     9*              NUMBER(I+1) = NUMBER(I+1)+NUMX
00122    10*              NUMBER(I) = NUMBER(I)-1000*NUMX
00123    11*     20       CONTINUE
00125    12*              IF (NUMBER(NSET+1).NE.0) NSET = NSET+1
00127    13*              MAKNXT = NUMBER(NSET)
```

```
00130    14*              RETURN
00131    15*              ENTRY NUM(I)
00133    16*              MAKNXT = NUMBER(I)
00134    17*              RETURN
00135    18*              END

      END OF UNIVAC 1108 FORTRAN V COMPILATION.     0 *DIAGNOSTIC* MESSAGE(S)
PHASE 1 TIME =    0.048 SEC.
PHASE 2 TIME =    0.033 SEC.
PHASE 3 TIME =    0.075 SEC.
PHASE 4 TIME =    0.038 SEC.
PHASE 5 TIME =    0.058 SEC.
PHASE 6 TIME =    0.055 SEC.

TOTAL COMPILATION TIME =    0.307 SEC.
```

TABLE OF FACTORIALS FOR 1 THROUGH 45

1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5,040
8!=40,320
9!=362,880
10!=3,628,800
11!=39,916,800
12!=479,001,600
13!=6,227,020,800
14!=87,178,291,200
15!=1,307,674,368,000
16!=20,922,789,888,000
17!=355,687,428,096,000
18!=6,402,373,705,728,000
19!=121,645,100,408,832,000
20!=2,432,902,008,176,640,000
21!=51,090,942,171,709,440,000
22!=1,124,000,727,777,607,680,000
23!=25,852,016,738,884,976,640,000
24!=620,448,401,733,239,439,360,000
25!=15,511,210,043,330,985,984,000,000
26!=403,291,461,126,605,635,584,000,000
27!=10,888,869,450,418,352,160,768,000,000
28!=304,888,344,611,713,860,501,504,000,000
29!=8,841,761,993,739,701,954,543,616,000,000
30!=265,252,859,812,191,058,636,308,480,000,000
31!=8,222,838,654,177,922,817,725,562,880,000,000
32!=263,130,836,933,693,530,167,218,012,160,000,000
33!=8,683,317,618,811,886,495,518,194,401,280,000,000
34!=295,232,799,039,604,140,847,618,609,643,520,000,000
35!=10,333,147,966,386,144,929,666,651,337,523,200,000,000
36!=371,993,326,789,901,217,467,999,448,150,835,200,000,000
37!=13,763,753,091,226,345,046,315,979,581,580,902,400,000,000
38!=523,022,617,466,601,111,760,007,224,100,074,291,200,000,000
39!=20,397,882,081,197,443,358,640,281,739,902,897,356,800,000,000
40!=815,915,283,247,897,734,345,611,269,596,115,894,272,000,000,000
41!=33,452,526,613,163,807,108,170,062,053,440,751,665,152,000,000,000
42!=1,405,006,117,752,879,898,543,142,606,244,511,569,936,384,000,000,000

```
43!=60,415,263,063,373,835,637,355,132,068,513,997,507,264,512,000,000,000
44!=2,658,271,574,788,448,768,043,625,811,014,615,890,319,638,528,000,000,000
45!=119,622,220,865,480,194,561,963,161,495,657,715,064,383,733,760,000,000,000

NORMAL TERMINATION AT LEVEL     0
LAST STATEMENT EXECUTED WAS     15 - MAIN

SNOBOL STATISTICS SUMMARY-

            213 MS. EXECUTION TIME
            665 STATEMENTS EXECUTED,        0 FAILED

* TIMER STATISTICS SUMMARY *

TIMING FOR  MAIN

 STATEMENT    # OF EXECUTIONS    TIME IN MILLISECONDS
      1                0                0.0
      2                0                0.0
      3                0                0.0
      4                0                0.0
      5                1                0.2
      6                1                0.0
      7                1                0.0
      8                1                1.0
      9                1                0.2
     10               45               15.4
     11               45                2.6
     12              340              132.6
     13              340               28.8
     14               45               27.8
     15               45                4.0
     16                0                0.0
     17                0                0.0
     18                0                0.0
```

REFERENCES

1. Farber, D. J., Griswold, R. E., and Polonsky, I. P., "SNOBOL, A String Manipulation Language," Journal of the ACM, 11, 1 (1964).

2. ..., ..., and ..., "The SNOBOL3 Programming Language," Bell System Technical Journal, 45, 6, (1966).

3. Forte, A., SNOBOL3 Primer, The M.I.T. Press, (1967).

4. Shapiro, M. D., "A Bibliography of SNOBOL Publications," SIGPLAN Notices of the ACM, 4, 11, (1969), pp. 41-48.

5. Griswold, R. E., Poage, J. F., and Polonsky, I. P., The SNOBOL4 Programming Language, Prentice-Hall, (1968), (First Edition).

6. Dewar, B. K., and Belcher, K., "SPITBOL," SIGPLAN Notices of the ACM, 4, 11, (1969), pp. 33-35.

7. Maurer, W. D., and Santos, P. J., "A SNOBOL4 Compiler," Fourth Annual Princeton Conference on Information Sciences and Systems, March 1970.

8. Santos,P. J., and Maurer, W. D., "Compilation of a Subset of SNOBOL4," SIGPLAN Notices of the ACM, 5, 12, (1970), pp. 60-68.

9. Knuth, D. E., The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, (1968), pp. 411-420, and 435-451.

10. Knowlton, K. C., "A Fast Storage Allocator," Communications of the ACM, 8, 10, (1965).

11. Morris, R., "Scatter Storage Techniques," Communications of the ACM, 11, 1, (1968).

12. Elson, M., and Rake, S. I., "Code-Generation Technique For Large-Language Compilers," I.B.M. Systems Journal, 9, 3, (1970).

13. Floyd, R., "Syntactic Analysis and Operator Precedence," Journal of the ACM, 10, July 1963, p. 316.