FASBOL II
A SNOBOL4 COMPILER FOR PDP-10

by

Paul Joseph Santos, Jr.

Memorandum No. ERL-M348

August 1972

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

FASBOL II

A SNOBOL4 COMPILER FOR THE PDP-10

Paul Joseph Santos, Jr.

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

ABSTRACT

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

While being almost totally compatible with SNOBOL4, Version 3, 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.

4. Capability of interfacing with FORTRAN and MACRO programs, providing any division of labor required by the nature of a problem.

5. Measurement and runtime parameter facilities to aid in optimizing execution time and/or storage utilization.

Research supported in part by the National Science Foundation, Grant GJ-821.

TABLE OF CONTENTS

CHAPTER 1

1. Introduction

The first FASBOL [1] was a similar system designed and written for the UNIVAC 1108 under the EXEC II operating system, and operational as of October, 1971. FASBOL II, the PDP-10 system, is an enhanced version which is in addition compatible with Version 3 of SNOBOL4. It is presumed that the reader is familiar with SNOBOL4, Version 3 as described by the second edition (1971) of the Prentice-Hall publication [2]. Using [2] as a base description of SNOBOL4, the following chapters explain any differences and additions present in FASBOL II, as well as describe how to use it to compile and run programs.

The FASBOL II compiler is itself written in FASBOL and is like FORTRAN and MACRO in that it accepts specifications for source, listing, and object files (the object is a MACRO program which must be assembled). The reason for writing the compiler in FASBOL was for speed of implementation, automatic checkout of the run-time library, and ease of modification. If after some use the compiler should prove to be unsatisfactory in terms of core utilization or execution speed, the MACRO stage can be hand-tailored, using the measurement techniques available in FASBOL, into a more efficient program. A further enhancement would be the direct production of relocatable code by a one-pass compiler written in either FASBOL or MACRO.

The FASBOL II run-time library is written in MACRO, since its efficiency is paramount, and is searched in library mode, after loading all FASBOL programs, in order to satisfy program references to

predefined primitives and system routines. Sections of the FORTRAN library may also be loaded, provided they do not compete with FASBOL for UUO's and traps.

Internal documentation of the operation of the run-time system is available as a separate document.

Use of the male gender in third-person references in this manual in no way implies that FASBOL is not useful for female persons; the author is simply not aware of any easy way to write in neuter.

CHAPTER 2

2. Language Description

The syntax for FASBOL II is given in Appendix 1. In addition to the detailed changes mentioned below, this syntax differs from that given in [2] only in the sense that it is more restrictive of compile-time syntax. For example, since FASBOL II does not permit redefinition of operators, the expression

```
( A . B ) + ( C . D )
```

is flagged as a compilation syntax error, whereas the interpreter (i.e. the system described in [2]) would accept it and then produce an "illegal type" error message during execution. Most SNOBOL4 programs should run "as is" under FASBOL II; Sections 2.1.1 and 2.1.2 describe exactly all features that may cause incompatibility, and the remaining sections deal with enhancements.

2.1 General language features

The following three sections discuss, respectively, features of SNOBOL4 not implemented in FASBOL II, features of SNOBOL4 implemented differently in FASBOL II, and additional features available in FASBOL II and described more completely in sections 2.2, 2.3, 2.4 and 2.5.

2.1.1 SNOBOL4 features not implemented

The predefined primitives EVAL and CODE, the datatypes CODE and EXPRESSION, and direct gotos (as used with CODE) are not implemented. They imply a run-time compilation capability which is not available in the FASBOL library at this time.

The redefinition of operators via OPSYN, or the redefinition of predefined primitive pattern variables (e.g. ARB) or functions (e.g. SPAN) is not permitted in FASBOL, which considers all these items as a structural part of the language essential to generating efficient code. For this reason, the keywords &ARB, &BAL, &FAIL, &FENCE, &REM and &SUCCEED are not needed and therefore not implemented. Also, &CODE has no meaning for the PDP-10, and is not available either.

The SNOBOL4 tracing capability is not implemented in FASBOL at this time. However, the &STNTRACE keyword (see section 2.5) provides some tracing capability.

Although QUICKSCAN mode for pattern matching is implemented in FASBOL, two features of this mode, available in SNOBOL4, are not implemented. There are a) continual comparison of the number of characters remaining in the subject string against the number of characters required by non-string-valued patterns, and b) assumption that unevaluated expressions must match at least one character. This implies that some matches may last a little longer and perhaps have a few more side-effects (e.g. via $), and that left-recursive pattern definitions will loop indefinitely (see Section 3.2.3).

2.1.2 SNOBOL4 features implemented differently

The FASBOL I/O structure is time-sharing oriented and does not use FORTRAN I/O, so that it differs somewhat from the SNOBOL4 I/O. Both input and output can be either in line or character mode. Line mode is similar to SNOBOL4 I/O, with input records being terminated just prior to a carriage return, line feed sequence, and with this sequence being

added to output records. Trailing blanks seldom occur on input, and so the &TRIM keyword is not implemented (but the TRIM function is). Character mode gets one character (including a carriage return or linefeed) on input, and outputs a string without sppending a carriage return, linefeed sequence to it. INPUT, INPUTC, OUTPUT and OUTPUTC have predefined associations (see 2.4.2, I/O primitives) corresponding to line and character mode teletype input and output, respectively. PUNCH does not have a predefined association. There are additional predefined primitives for device and file selection, etc., discussed in Section 2.4.2.

Changes in program syntax are as follows:

a) Compiler generated statement numbers are always on the left.

b) Source lines are always truncated after 72 characters.

c) The character codes and extended syntax are like the S/360 version, except the character ! (exclamation point) replaces | (vertical stroke) and \ (back slash) replaces ¬ (not sign).

d) Binary $ and . (immediate and conditional pattern assignment) have lower precedence than the binary arithmetic operators, but higher precedence than concatenation. Thus, the expression

```
X A + B $ C
```

is taken to mean

```
X ((A + B) $ C)
```

In SNOBOL4, $ and . have the highest precedence of all binary operators, and would give the meaning

```
X (A + (B $ C))
```

to the above expression.

Changes in program semantics and operation are as follows:

a) The binary . (name) operator always returns a value of type NAME (SNOBOL4 sometimes returns a STRING). Indirection (unary $) applied to a value of type NAME returns the same as value. Names of TABLE entries are permitted.

b) Some predefined primitive functions operate differently than in SNOBOL4 (see Section 2.4.2).

c) &MAXLNGTH is initially set to 262143 (in SNOBOL4, the value is 5000). This value is also the absolute upper limit on string size.

d) If &ABEND is nonzero at program termination, an abnormal (EXIT 1,) exit to the system is taken.

e) Primitive functions may be called with either too few or too many arguments, even via OPSYN or APPLY.

2.1.3 Additions to SNOBOL4

Declarations are provided in FASBOL for the enhancement of programs. No declarations are ever required, but if they are used they must all precede the first executable statement in a program. Declarations are described in Section 2.2.

Additional control cards and compilation features are available, discussed in Section 2.3, and additional predefined primitive functions and keywords are described respectively in Sections 2.4 and 2.5.

Additions to program syntax are as follows:

a) Quoted strings may be continued onto a new line, with the continuation character removed from the literal.

b) Single and double quotes may be included in a literal that is bracketed by the same, by use of the construction '' to stand for ' and "" to stand for " inside of literals bracketed by ' and ", respectively.

c) Comment and control lines (i.e. starting with * or -) may start inside a line image (i.e. after a ;), and consume the remainder of the line image.

d) The run-time syntax for DEFINE and DATA prototypes has been loosened to conform with the rest of FASBOL syntax by permitting blanks and tabs after ((open parenthesis), around , (comma), and before) (close parenthesis).

2.2 Declarations

FASBOL declarations have two primary purposes. One purpose is to optimize a program in space and/or time. The second purpose is to allow inter-program linkage and communication. The general form of a declaration is a call on the pseudofunction DECLARE, with two or three arguments, the first of which is always a string literal identifying the type of declaration, and the remaining arguments specifying the parameters or program symbols upon which the declaration has effect. As has been noted, all declarations must precede the first executable statement; this also implies no declaration line may contain a label. A FASBOL program with declarations can be made otherwise compatible with a SNOBOL4 interpreter by inserting the statement

```
DEFINE('DECLARE()', 'RETURN')
```

at the beginning of the program.

2.2.1 PURGE and UNPURGE

Normally, a FASBOL application will involve a main program and several independently compiled subroutines. During execution, the run-time system maintains a run-time symbol table for each separately compiled program, as well as a global symbol table. In the absence of declarations to the contrary, all explicitly mentioned variables, labels, and functions are put into the local symbol table for that program. Thus, program X and program Y may both have labels LAB to which they perform indirect gotos. The global symbol table contains such global symbols as OUTPUT and RETURN, and any new symbols that arise during execution of any of the programs. A symbol lookup in program X first searches the local symbol table for program X, then the global symbol table, and then, if still not found, creates a new entry in the global symbol table. Thus a local symbol table never grows beyond the size determined for it at compilation time.

The purpose of the PURGE.VARIABLE, PURGE.LABEL, and PURGE.FUNCTION declarations is to eliminate symbols from the local symbol table and thus conserve space. This can be safely done for labels provided that the label is never referenced indirectly ($ goto), or explicitly and/or implicitly in a DEFINE call. A similar criterion applies to safely eliminating variables, only the number of cases to watch for is greater; any situation that requires an association between the string representing

symbol table instead of the local one. Only one subprogram may globalize a particular symbol, since the implication is that the variable, label, or function belongs to that program. Any other program that does not have a similar symbol in its local symbol table will then be able to reference the global symbol.

While GLOBAL provides for interprogram communication via the symbol table, the ENTRY/EXTERNAL declarations provide for more direct interprogram communication by using the linking loader to connect external references. The ENTRY.VARIABLE, ENTRY.LABEL, and ENTRY.FUNCTION declarations make the specified local entities accessible to external programs. The second and third arguments to the ENTRY.FUNCTION declaration are like the arguments to DEFINE, and the function is automatically DEFINED the first time it is called, so no extra DEFINE is necessary. The ENTRY.FORTRAN.FUNCTION declaration is similar to ENTRY.FUNCTION except that the compiler assumes the entry will be called by a FORTRAN program. Any combination of FASBOL, FORTRAN, and MACRO programs is permitted, provided the main program is FASBOL, and certain restrictions on FORTRAN code (see Section 3.2.5) are observed.

The EXTERNAL.VARIABLE, EXTERNAL.LABEL, EXTERNAL.FUNCTION, and EXTERNAL.FORTRAN.FUNCTION declarations are the converse of the ENTRY declarations, and imply that the specified entities be outside the program. The EXTERNAL.FORTRAN.FUNCTION declaration has a special form for its parameter list (second argument) where the number of arguments expected by each function is given in parentheses. The function value type is either declared implicitly (first character = I,J,K,L,M,N means

the variable name, and the actual location assigned to that variable, is such a case. For example, the statement

```
INPUT( 'VARB',0,60)
```

implies that the variable VARB, if it is mentioned explicitly in the program and thus assigned a location, must be in the run-time symbol table. An explicit reference to VARB would be, for example,

```
TTYLIN   =   VARB
```

On the other hand, a variable that is never referenced explicitly need not be in the local symbol table, but the first symbol lookup for it will create an entry for it in the global symbol table. In the case of functions, the only symbols that can be safely purged are the ones corresponding to predefined primitives, since all others are needed to be able to define the user functions, via DEFINE or otherwise.

When there appear to be more symbols of a given type to be purged than left in the symbol table, the second argument to the declaration can be the pseudovariable ALL; then, the UNPURGE.VARIABLE, UNPURGE.LABEL or UNPURGE.FUNCTION declarations can be used to place specific symbols into the symbol table.

2.2.2 GLOBAL, ENTRY, and EXTERNAL

These declarations permit interprogram communication on an indirect (i.e. symbol lookup) and/or direct (i.e. loader linking) basis. The GLOBAL.LABEL, GLOBAL.VARIABLE, and GLOBAL.FUNCTION declarations override PURGE/UNPURGE and cause the specified symbols to be placed in the global

integer, otherwise real), or explicitly by appending =INTEGER or =REAL to the function value, before the argument specification.

2.2.3 Type declarations

Normally, FASBOL variables may contain data of any type, referred to here as descriptor mode variables. Sometimes it is known in advance that certain variables will always take on values that are integers or reals; in this case it becomes advantageous to declare them INTEGER or REAL. In addition to the execution speed advantages (see Section 3.2.1), they can be passed to FORTRAN directly in function calls. Also, the only way to pass a string (other than a literal) to FORTRAN is to use a variable declared to be STRING, which is the only real use for that declaration; a fixed amount of storage is allocated for the variable based on the max character count given in parenthesis after each name in the parameter list (second argument). The only restrictions on these variables, referred to here as dedicated mode variables, is that they may not have I/O associations. All keywords (except for &RTNTYPE and &ALPHABET) are treated as dedicated integers.

2.2.4 Other declarations

The OPTION declaration serves to specify various compilation options. The HASHSIZE=n declaration, ignored in all but the main program, is used to cause a larger or smaller than normal hash bucket table to be allocated for use by the run-time symbol table. The number n should be a prime and represents the number of buckets in a linked hash table; the standard value is 127. This bucket table is at the center of all symbol lookups in the runtime system, including TABLE references, so that there is a distinct tradeoff between the

sparsity of the table and the time required for a lookup. The NO.STNO option causes the compiler to eliminate the normal bookkeeping on &STNO, &STCOUNT, etc. that occurs each time a statement is entered, and is helpful to speed up slightly the execution of debugged programs. The TIMER option, which is incompatible with NO.STNO in the same program, adds to the normal bookkeeping a valuable statement timing feature (see Section 3.2.4). The timing statistics on each program being timed are printed out at the end of execution, and intermediate timing statistics can be printed out during execution by using the primitive EXTIME (see Section 2.4.2).

The SNOBOL.MAIN and SNOBOL.SUBPROGRAM declarations indicate whether the program is a main program or a subprogram, and gives it a name (i.e. TITLE in MACRO). In the absence of either declaration,

```
DECLARE( 'SNOBOL.MAIN', '.MAIN.')
```

is assumed.

The RENAME declaration is used primarily to rename predefined symbols (see Appendix 2) that would otherwise conflict with a given user's. For example, if a user wished to have a variable called ARB, or his own IDENT function while retaining the primitive also, he should rename them some other names. On the other hand, if a user wants to re-define IDENT, for example, no RENAME should be used, and IDENT will become redefined when his own DEFINE is executed.

Although usually the order in which declarations occur is not important, all PURGE.X, ALL declarations should precede others which

also refer to entities desired to be purged. For example, the sequence

```
DECLARE('ENTRY.VARIABLE', 'A,B,C')
DECLARE('PURGE.VARIABLE', ALL)
```

will cause the variables A, B and C to be missed and included in the symbol table, since the purge flag only has effect on new symbols.

It should also be noted that whereas the syntax of variable and function name lists uses a comma as a separator, label lists are separated by blanks. The reason for this is that the syntax for labels includes commas, but a blank is a valid label terminator. Also, all quoted strings in declarations are delimited by single quotes. A single quote may be entered inside such a string for example, in a label) by using the '' convention mentioned in Section 2.1.3.

2.3 Control

In FASBOL there is an expanded repertoire of control cards for controlling listing, cross-referencing, and failure protection. In the following list, the first of a pair controlling a switch is the initial mode.

LIST, UNLIST	turns program listing on, off
NOCODE, CODE	turns object listing off, on (the generation of object code can be inhibited by not specifying an object)
EJECT	causes a page eject (form feed)
SPACE n	spaces n lines (or 1 line if n is absent)
NOCROSS, CROSREF	turns symbol cross-referencing off, on. This can be done for a whole program or selectively for parts of it

FAIL, NOFAIL* turn off, on a compiler feature that 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. An unconditional GOTO is equivalent to none at all, and will be trapped if the statement fails.

2.4 Predefined primitives

In the following sections, only those primitives which differ from SNOBOL4 or are new in FASBOL will be discussed. Appendix 2 has a complete list of primitives available in FASBOL.

2.4.1 Pattern primitives

Three new primitives in the SPAN/BREAK class have been added; these are structural, like the other pattern primitives, and cannot be redefined.

NSPAN(class) is like SPAN, but may match the null string.

BREAKQ(class) is like BREAK, but does not look for break characters inside of substrings delimited by single (') or double (") quotes.

BREAKX(class) is like BREAK, but has alternatives that extend the match up to each succeeding break character. Operates like

```
BREAK(class)  ARBNO(LEN(1)  BREAK(class))
```

2.4.2 Expression primitives

A number of SNOBOL4 primitives work somewhat differently in FASBOL, and new primitives have been added for I/O, string manipulation, and communication with the run-time system.

---

*Credit for this idea goes to the authors of SPITBOL [3], who also inspired the inclusion of DUPL, LPAD, RPAD, and REVERS.

COLLECT(n) forces a garbage collection, returns the total number of words collected, and fails if no block of size n or larger is available.

CONVERT(table, 'ARRAY') and CONVERT(array, 'TABLE') are implemented differently, by removing the above facilities from CONVERT and putting them in ARRAY and TABLE. See below.

ARRAY(table) converts a TABLE datatype to an ARRAY as described in [2], pp. 122. An empty TABLE causes ARRAY to fail.

TABLE(array) converts certain types of ARRAY datatypes to a TABLE as described in [2], pg. 122. The TABLE datatype is different from all others in that, once it has been created, it exists independently from its use in the program. Thus, to reclaim the storage, it must be explicitly deleted by TABLE(table). Once a table has been deleted, further references to it are illegal.

APPLY(fun, args) will accept either more or fewer arguments than required by the function; it will reject extra ones or fill in missing arguments with null values.

OPEN(device, chan) opens an I/O device on a software channel, assigns buffers, and returns the channel number.

If chan < 0 or > 15, illegal I/O unit error.

If chan = 0, an unused channel is assigned; if channel table is full (> 15 channels), error.

If chan is already in use, illegal I/O unit error.

If device is not a string of the form:
devnam [([outbuf] [, [inbuf]])] ,
it is a bad prototype or illegal arg error.

If devnam is not recognized, or is not a file structure and is already assigned to a channel, illegal I/O unit error.

If ([outbuf] [, [inbuf]]) is missing, (2,2) is assumed. If either outbuf and/or inbuf is missing, 0 is assumed for the missing value.

If the device allows only input (or output), the other buffer parameter is ignored.

Examples:

```
OPEN('DTA3(,4)', 5)
OUTPUT('DUMP', OPEN('MTA0'), 1000)
```

RELEASE(chan) releases the software channel and all associations to it, returns all buffers to free storage, and returns a null value.

If chan < 0 or > 15, illegal I/O unit error.

If chan = 0, release all channels in use.

If channel not in use, ignore and return.

LOOKUP(file, chan) opens file for input (reading) on software channel, returns channel.

ENTER(file, chan) opens file for output (writing) on software channel, returns channel.

If chan < 0 or > 15, illegal I/O unit error.

If chan = 0, a preliminary OPEN('DSK') is performed, the new channel returned.

If file is not of the form
filnam [. ext]
, bad prototype error.

If file is not found, or if channel is not open for operation, illegal I/O unit error.

If input (LOOKUP) or output (ENTER) side of channel already selects a file, the old file is closed.

CLOSE(chan, inhib, outhib) closes the input and/or output side of the software channel, returns null.

If outhib is non-null, the output side is not closed.

If inhib is non-null, the input side is not closed.

If chan < 1 or > 15, illegal I/O unit error.

If channel is not in use, ignore and return.

INPUT(var, chan, len)
OUTPUT(var, chan, len) create an input(output) association between the variable var and software channel chan, with line/character mode and association length specified by len, and return null.

If len > 0, line mode.

If len = 0, line mode with default length (72).

If len < 0 or not INTEGER, character mode.

If chan > 15, illegal I/O unit.

If chan > 0 use channel table to determine I/O device.

If chan = 0 use TTY I/O

If chan < 0 or not INTEGER, disconnect association but do not DETACH it.

If the input (output) side of the channel has not been opened, illegal I/O unit error.

If var is not a string, illegal arg.

If an association for var already exists, it is changed.

If variable is dedicated, illegal arg.

Examples:

```
INPUT('SOURCE',LOOKUP('SRCELT.SNO'),80)
OUTPUT('TYPEOUT.CHARS',0,-1)
```

The initial I/O configuration is equivalent to:

```
INPUT('INPUT')
INPUT('INPUTC',0,-1)
OUTPUT('OUTPUT')
OUTPUT('OUTPUTC',0,-1)
```

During execution, all system messages are output via the variables OUTPUT and OUTPUTC (which should always both be associated to the same channel). In order to switch system output to the printer, for example,

```
LPT = OPEN('LPT')
OUTPUT('OUTPUT',LPT,132)
OUTPUT('OUTPUTC',LPT,-1)
```

Channel 0 is never assigned, but when used in INPUT and OUTPUT associations implies the user TTY and TTCALL operation. On input, line mode reads up to (but not including) the next carriage return, line feed (CR,LF) sequence, and then these are discarded. Character mode reads only one character (including CR or LF). Line mode discards any characters beyond the association length. An EOF causes failure in either mode, but cannot occur on some devices (such as the user TTY).

On output, line mode writes out the string value with a CR, LF appended, whereas character mode does not append the CR, LF. In line mode, if the string length is greater than the association length, extra CR, LF characters are inserted every association length substring.

DETACH(var) disconnects input and output associations for the variable and detaches it from I/O processing, returns null.

If the variable had no association, ignore and return.

SUBSTR(string, len, pos) returns the substring of string starting at pos of length len, and fails if len < 0, pos < 0, or pos + len > SIZE(string). The position convention is the same as that for patterns, and the operation is similar (but faster and less space-consuming) to:

```
string  TAB(pos)  LEN(len) . SUBSTR
```

INSERT(substring, string, len, pos) returns the string formed by substituting for the one specified by the last 3 arguments into string, failing under that same conditions as:

```
string  TAB(pos) . PART1   LEN(len)   REM . PART2
INSERT = PART1 substring PART2
```

LPAD(string, len, padchr) returns the string formed by padding string on the left with padchr characters to a length of len. If string is already too long, it is returned unchanged; if padchr has more than one character, only the first is used. If the third argument is null, blanks are used

RPAD(string, len, padchr) is like LPAD, but pads to the right.

REVERS(string) returns the string formed by reversing the order of the characters of its argument.

EXTIME(progname) causes the runtime system to output current timing statistics for the program progname and returns null, or fails if the program is not being timed.

REAL(x) is like INTEGER for reals

EJECT() causes a page eject (form feed) to be assigned to OUTPUTC

DAYTIM() returns an 11-character string representing the time of day (since midnight), as

```
HH:MM:SS.HH
```

meaning hours, minutes, and seconds to the nearest hundredth.

2.4.3 FORTRAN primitives

These are predefined EXTERNAL.FORTRAN functions that, except for FREEZE, merely perform some simple arithmetic task and have integer values.

FREEZE() can be called to freeze the state of the FASBOL execution for resumption at some future date. When FREEZE is called, it exits to the monitor; the job may be SAVED, and when run again, it will start off by returning from the call to FREEZE. This is particularly useful for some applications that perform a considerable amount of initialization and wish to be able to start after that point on a repeated basis. No I/O devices (other than the console TTY) may be open at the time of the FREEZE and the call should be made from function level 0 if any timing is active.

ILT(int, int) or ILT(real, real)
[also ILE, IEQ, INE, IGE, IGT] Like LT, etc. except more efficient for dedicated variables or expressions.

AND(int, int)
[also OR, XOR, RSHIFT, LSHIFT, REMDR] perform the specified arithmetic or logical operation on their integer arguments and return the value (logical AND; inclusive OR; exclusive OR; logical right and left shift of first by second argument; remainder of integer division of first by second argument).

NOT(int) returns one's complement of its argument.

2.5 Keywords

Three new keywords, all unprotected, have been added in FASBOL.

&STNTRACE is initially 0, but if assigned a nonzero value it causes a trace output for each statement, giving statement number, program name, and time. This slows down execution considerably, so it is best to turn it on as close to the suspected bug as possible. Programs compiled under the NO.STNO option will ignore the value of &STNTRACE, however, so another approach is to run will all but the suspect program under NO.STNO, with &STNTRACE on all the time.

&DENSITY is initially 75, and represents the desired density of free storage immediately following a garbage collection. For example, &DENSITY = 75 means that the free storage system will try to maintain at least a 1:4 ratio between available and total storage immediately following a garbage collection, and will expand total storage as far as necessary or possible in order to try to maintain this ratio. See Section 3.2.4.

&SLOWFRAG is initially 0, but if assigned a nonzero value it serves to switch in a heuristic in the free storage mechanism that slows down the rate of fragmentation of blocks at the expense of some wasted storage. See Section 3.2.4.

CHAPTER 3

3. FASBOL II Programming

Using FASBOL involves two separate stages, as in FORTRAN: compilation and execution. The first requires the compiler, an absolute program named FASBOL.SAV (or FASBOL.DMP, depending on the operating system). Execution of compiled (and then assembled) programs requires a library search, during loading, of the FASBOL library; this is a collection of relocatable programs named FASLIB.REL. The relative accessibility of these programs will depend on the installation. The compiler requires a minimum of 35K to run, and requires more core in proportion to the program being compiled. The core requirements for execution of user programs depends on the size of the compiled programs plus at most 5K (if every single facility in the library is used) for the library.

3.1 Using the compiler and runtime library

To compile a FASBOL program, type

.RUN FASBOL n

where the CORE argument (n) is optional. It is best to give the compiler an amount of core commensurate with the size of program being compiled; this will increase compilation speed by minimizing garbage collections, since the compiler will expand core on its own only when it absolutely has to.

The compiler will respond with

*

, to which the user is expected to respond with a set of file specifications of the form

*macfil,lstfil←srcfil

Each file specification is of the standard form, as would be given to MACRO, for instance. The MACRO output file, macfil, is given a default version of MAC if not specified. Lstfil is the listing file, and both macfil and lstfil are optional. The source file, srcfil, is given a default version of SNO if not specified. Only one source file is permitted.

Examples:

*DTA3:SAMPLE,LPT:←SAMPLE

*,TAPE.LST←MTAØ:

*NEW.NEW,←TEST.NEW

Once the compiler produces the MACRO output file, it must be assembled, using the Q flag to suppress anxiety messages from MACRO:

.COMPILE macfil(Q)

The MACRO file can be deleted after assembly, as it will be of little interest to most users; it is mainly a shortcut for the compiler to avoid having to generate relocatable code. On the other hand, those individuals who understand the workings of the run-time system may wish to hand-tailor these intermediate programs to suit their own needs; Caveat Emptor.

To prepare any collection of FASBOL and other programs for execution, the command list should be terminated with a library search of FASLIB, for example:

.LOAD fil1,fil2, . . .,filn,FASLIB/LIB

It is important that FASLIB be searched only once, after all FASBOL programs have been loaded, since it is very carefully sequenced to provide dummy versions of elements that are somehow referenced, but not really needed. The automatic search of the FORTRAN library should take place after

searching FASLIB, since FASLIB may require some FORTRAN routines.

While FASBOL may call or may be called by FORTRAN or user MACRO programs, the main program must be a FASBOL program. Furthermore, the FASBOL runtime system enables traps and uses user UUO's 1 through 10, so it is incompatible with the FORTRAN runtime system. What this means is that FORTRAN programs used within a FASBOL execution must not do any I/O or otherwise cause FORSE. to be loaded. FASBOL does provide an infinite stack (all FASBOL stacks are infinite, up to user core limits) in register 17, however, so a broad class of FORTRAN user programs and library routines are permissible.

Unless changed by the user's program, all system output during execution is sent to the user's console; upon either error or normal termination of execution, the appropriate messages and statistics will be printed out, and control returned to the monitor. The error numbers are described in Appendix 3.

3.2 Programming techniques

Because of the basic difference between interpretive and compiler systems, and the additional features available in FASBOL, some programming techniques besides those discussed in [2], Ch. 11, are described here. An interested user may wish to get a listing of the compiler itself to see examples of some of these techniques.

3.2.1 Dedicated expressions

Dedicated expressions in FASBOL are those that are known, because of some component, to have a numerical value of a predetermined type. At one extreme is the totally dedicated statement that involves nothing but declared dedicated variables, constants, and perhaps FORTRAN calls.

For example, if I were declared INTEGER, the statement

```
I = 2 * I + 10
```

would be totally dedicated, and compile into

```
MOVE    1, I
IMULI   1, 2
ADDI    1, 10
MOVEM   1, I
```

Even if an expression is mixed, with both dedicated and descriptor-mode subexpressions, in-line arithmetic code is compiled for as much of the expression as is possible to commit to a specific type of value (i.e. INTEGER or REAL) at compile time. It is therefore to the user's advantage to declare as many variables as he perceives will be dedicated in use to be of that dedicated type. Not only will the program run faster, it may even use less core. In a situation where all entities are descriptor mode, even arithmetic operators have to check the type of, and possibly convert each argument.

In this connection it should also be noted that the predefined FORTRAN primitives ILT, ILE, IEQ, INE, IGE, IGT have been provided in order to do a much more efficient job than LT, ..., GT when the arguments are dedicated. For example, if R and S are REAL, the test

```
ILT(R, S)
```

takes up several fewer words and runs about 100 times faster than the test

```
LT(R, S)
```

FASBOL permits mixed mode (INTEGER and REAL) arithmetic, the general rule being that the result of an operation is INTEGER only if both sides are integer; furthermore, an arithmetic operation involving dedicated and descriptor mode values always has a dedicated result. A value being combined with a stronger mode is first converted, and then the operation is

performed in that mode; for example, if I is INTEGER but D has not been declared dedicated,

```
I + D
```

implies the value of D will first be converted to an integer, and then added to I. The only exception to this rule is the ** (exponentiation) operator, which permits a REAL raised to an INTEGER power.

Finally, it should be noted that whereas the range of values of dedicated variables is the same as in FORTRAN, descriptor mode integers have a range two powers of 2 less in magnitude, and descriptor mode reals have two fewer bits of precision in the mantissa. The reason for this is that the two bits are needed for the descriptor type.

3.2.2 Use of the unary ? and . operators

Unary ? (interrogation) is useful to indicate to the compiler that an expression is evaluated for its effect, rather than value. For example, a frequent occurrence in SNOBOL programs is the concatenation of null-valued functions for their sequential effects and/or succeed/fail potential. If the compiler knows that an element has a null value, it does not generate code to include it in the concatenation. Therefore it is efficient to precede predicates and other null-valued elements in a concatenation with the ? operator. This technique is especially valuable when combining predicates and dedicated arithmetic, as in

```
I = ?IDENT(A, B)  ?IGT(I, 25)  I + 1
```

, since concatenation is avoided entirely and the dedicated arithmetic is performed after the execution of the predicates without any need for conversion between dedicated and descriptor values.

Another frequent occurrence in SNOBOL programs is the repetitive

access of the same indirect variable, array element, or field of a programmer-defined datatype. Each of these accesses, whether to retrieve or store a value, involve some overhead which is repeated for each access. For example, in the statements

```
$X  =  $X + 1
```

, the variable represented by the string value of X is looked up in the symbol table twice, the first time to retrieve its value, the second time to store into it. The unary . (name) operator can be used to save the result of one lookup by creating a NAME datatype, and then the NAME can be used in an indirect reference wherever the original expression was used. Instead of the above statement, a more efficient sequence would be

```
 Z  =  .$X
$Z  =  $Z + 1
```

, where Z contains the NAME of the variable pointed to by X. The same considerations apply to array references and field references as apply to indirection; it is efficient to save the NAME of the variable referenced if it will be used more than once in close successsion. For example, the statements

```
A<25>        =  F(A<25>)
NEXT(LIST)   =  NODE(VAL, NEXT(LIST))
```

would be more efficient if coded as

```
 Z  =  .A<25>
$Z  =  F($Z)
 Z  =  .NEXT(LIST)
$Z  =  NODE(VAL, $Z)
```

It should be noted that TABLE references are array references, and the symbol lookup involved makes it even more efficient to save the NAME. The NAME of an ARRAY or TABLE element, or of a field of a programmer-defined datatype, is only valid as long as that array, table or datatype exist;

attempts to retrieve or store using the NAME afterwards will have unpredictable results. Also, the NAME of a variable, evaluated before that variable acquires an I/O association, does not reflect that association.

3.2.3 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 best be achieved by enclosing the concatenation in parenthesis, and, in this case, using the ? operator

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

, which will suppress string concatenation.

One particularly unique feature of FASBOL is that explicit pattern expressions, i.e. those involving the pattern operators and/or primitives, are compiled as re-entrant subroutines, rather than constructed at run-time into intermediate-language structures. The significance of this to the programmer, aside from the increase in execution speed, is that there is less of a need to pre-assign subpatterns that will appear in pattern matches later on; in fact, an unnecessary pre-assignment will be slightly less efficient because the pattern match will have to recurse one level deeper than otherwise during execution. The way to determine the need for pre-assignment is to note how much evaluation is actually required in a subpattern; if little or none is required, it can just as efficiently be included in the body of the match. Of course, if a subpattern is large

and/or used in several matches, the programmer may wish to pre-assign it anyway for convenience sake. Pattern evaluation involves only the elements of the pattern, not the structure itself. Literals and other constant values do not require evaluation, so the pattern

```
TAB(7)  (SPAN('XYZ') ! BREAK(';'))  $ SYM ';??'
```

requires no evaluation at all. A generally applicable rule for all FASBOL programming is that it is more efficient to use, wherever possible, literals instead of variables with constant value. Simple variables appearing in a pattern require little evaluation (only a determination if they have a string or pattern value), and even character class primitives (i.e. SPAN, BREAK, etc.) require little evaluation, if their argument is non-literal, provided the argument is a variable with a constant value. Examples of pattern elements requiring more extensive evaluation are (non-pattern primitive) function calls, non-pattern expressions requiring considerable evaluation in their own right, and character-class primitives whose arguments are other than literals or simple variables. An example of the latter case would be

```
ANY('XYZ' OTHER)
```

; even if the value of OTHER remains constant, the concatenation produces a new string each time, which prevents ANY from immediately using the break table it has generated on the last execution of that call. It has been assumed in all this discussion of pattern evaluation that the value of the pattern element would not change value between the time of assignment of the subpattern and its use in a match. Should this not be the case, of course, the alternative of including the subpattern in the match does not exist.

Even when integer constants cannot be used, it is still helpful to

use dedicated integer variables or expressions in patterns, if possible. Dedicated integer expressions are ideally suitable as arguments to the positional pattern primitives (i.e. POS, LEN, etc.), and integer variables are ideally suitable as objects of the cursor assignment operator (@). For example, suppose one wishes to take a string composed of sentences separated by semicolons (and terminated by a semicolon) and output the sentences on separate lines. A singe pattern match to do this would be (P is INTEGER):

```
   STRING @P SUCCEED TAB(*P)  BREAK(';')  $ OUTPUT  LEN(1)
.  @P  RPOS(0)
```

Note that the pattern requires no evaluation.

Since FASBOL does not employ the QUICKSCAN heuristic of assuming at least one character for unevaluated expressions, left-recursive patterns will loop indefinitely 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. To take a simple example, the pattern

```
P   =   *P  'Z'  !  'Y'
```

, which 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
```

3.2.4 Timing and storage management

The TIMER option permits the programmer to monitor the operation of any (or all) separately compiled programs, and provides feedback on where the time is being spent. Initial programming of some problem can be done rapidly with not much attention being paid to optimization. It is usually

the case that some small sections of a program account for a large percentage of the execution time; these are identified using the TIMER option. The programmer's time is then spent most efficiently optimizing the critical areas and ignoring the rest. Of course, after a series of optimizations, a new bottleneck will develop; the process can then be iterated until the law of diminishing returns takes hold. Finally, the TIMER declarations can be removed and the programs run in production mode.

The programmer has a large degree of control over storage management in FASBOL, which in turn means control over the space/time tradeoff that exists due to the dynamic storage allocation system (free storage). To begin with, requests from the free storage system prior to the first garbage collection (regeneration of dynamic storage) have very little overhead compared to ones subsequent to the first garbage collection. Unless there are good reasons for the contrary, the user should capitalize on this by starting his execution with approximately the amount of core he expects will eventually be required - past experience with the program is the best guide. Thus the number of garbage collections will be reduced to a minimum, and initial execution speeded up. In the absence of a core specification, the program will begin with the minimum required for loading, and will expand core as it becomes necessary, but undergoing more garbage collections.

The &DENSITY keyword is also useful in controlling the space/time tradeoff. &DENSITY may be set dynamically to any value between 1 and 100; immediately following a garbage collection, the dynamic storage allocation mechanism attempts to satisfy this value, interpreted as the percentage of total storage allocated that is in use at that time. Nothing is done unless

the actual ratio is greater than the desired one, in which case core is expanded to satisfy the desired ratio, or until user core limits are reached. For example, a user who sets &DENSITY to 99 is saying he wishes to keep his core size to a minimum, and is willing to pay a (rather large) premium in repeated garbage collections. On the other hand, a user who sets &DENSITY to 1 is asking for all the core he can get, in order that his program execute as rapidly as possible. It is also perfectly feasible to use a strategy where &DENSITY is set to different values at different times during execution. The initial value of &DENSITY is 75, which represents a general-purpose compromise.

If a user's application will occasionally require large contiguous blocks of storage, he may give himself 100% insurance by reserving dummy arrays of the appropriate size at the very beginning of his program. An alternative is to turn on the keyword &SLOWFRAG, which activates a heuristic which tends to slow down the fragmentation of large blocks at the expense of some wasted storage. While not 100% guaranteed, it will give the desired effect in most cases, minimizing the situation where a large block is called for, and though enough total storage is available, no contiguous area is large enough to satisfy the request.

Finally, the COLLECT primitive may be invoked at appropriate times, both to force a regeneration and also measure the amount of storage that is available.

3.2.5 FORTRAN interface

External FORTRAN subroutines, whether user-written or from the FORTRAN library, must be declared, including the number of arguments expected. A function call to the external subroutine may have any expression (except

patterns) as arguments, but all but a few recognized expressions are assumed to evaluate to integers and will cause an error exit if not. Dedicated integer and real variables are passed directly to the subroutine, as they would in a call from a FORTRAN program. Also, dedicated integer and real expressions are evaluated, the value saved in a temporary location, and this location passed to the FORTRAN routine. Finally, dedicated string variables and literals are passed to FORTRAN as a vector, the first word of which is pointed to by the calling sequence, and the FORTRAN routine may interpret it as a one-dimensional array of ASCII characters packed five to a word. In addition to returning an integer or real value, the function may modify the value of any dedicated integer, real, or string variable that is passed to it. In the last case, not only may the characters be modified, but the character count may be changed by storing it in the right half of the word immediately preceding the first word of the string (array (0)). For example, suppose a FASBOL program contains the declarations

```
DECLARE('INTEGER' , 'I')
DECLARE('REAL' , 'R')
DECLARE('STRING', 'S(15)')
DECLARE('EXTERNAL.FORTRAN.FUNCTION' , 'GETDAT(3)')
```

and the FORTRAN function GETDAT is defined as

```
FUNCTION      GETDAT(INDEX, ISTR, IDAT)
COMMON        IDATA(1000, 4)
EQUIVALENCE   (RDATA, IDATA)
DIMENSION     ISTR(3), RDATA(1000, 4)
ISTR(1)    =  IDATA(INDEX, 1)
ISTR(2)    =  IDATA(INDEX, 2)
ISTR(0)    =  (ISTR(0) / 2**18)*2**18 + 10
IDAT       =  IDATA(INDEX, 3)
GETDAT     =  RDATA(INDEX, 4)
RETURN
END
```

, then a typical use of GETDAT within the FASBOL program might be

```
R  =  GETDAT(2, S, I)
```

, which would have the effect of setting I to some integer value, R to some real value, and S to a 10-character string.

Entries that are expected from FORTRAN must be declared with the ENTRY.FORTRAN.FUNCTION declaration. This works like ENTRY.FUNCTION in that an automatic DEFINE is performed on the first call. Valid actual arguments in the FORTRAN call to FASBOL can be integers, reals, and Hollerith arrays (as described above, denoted by the codes 0, 2, and 5 in the calling sequence. Upon entering FASBOL, the actual arguments are copied, and converted if necessary, into the formal arguments, which are dedicated or descriptor mode FASBOL variables. The right half of the word immediately preceding the first word of a Hollerith array is considered to be the character count, and may be modified by FASBOL if the string argument is modified. Upon return to FORTRAN (via RETURN), the function value is determined by dedicated or descriptor value in the variable corresponding to the function name, but must be integer or real. The formal argument values are copied back (and re-converted, if necessary) into the actual arguments in the FORTRAN calling sequence, thus providing a means of passing back additional values, besides the function value, to FORTRAN.

It should be noted that FORTRAN is not recursive, and therefore any recursive combination of FASBOL and FORTRAN calls will not work. Even when a series of calls is not recursive, care must be taken not to re-enter a FASBOL routine which has a FORTRAN call pending, because the FASBOL routine uses the same temporary storage locations for all FORTRAN calls, including the predefined primitives IGT, etc.

In writing FORTRAN programs to be used with FASBOL programs, care should be taken not to perform any I/O, or use any other FORTRAN facility that requires the FORTRAN runtime system (FORSE.) to be loaded.

APPENDIX 1

Syntax for FASBOL II

Explanation of syntax notation

1. All terminal symbols are underlined, the remainder of the syntax consisting of non-terminals and syntax punctuation

2. The ::= operator indicates equivalence

3. The | operator indicates a series of alternatives

4. The blanks between consecutive elements indicate concatenation

5. The \ operator idicates the specific ruling out of the immediately following element as a precondition for further concatenation

6. The ... operator indicates the indefinite repetition of the immediately preceding element

7. The < > brackets serve to group expressions into a single element

8. The [ ] brackets indicate the optional occurrence of the expression contained within the brackets, and also serve to group the expression into a single element

9. The order of precedence for the operators, from highest to lowest, is:
   \ ... (blanks) | ::=

Syntax

```
program::= [declaration | comment]... [execute.body] end.statement

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 | tab> [bl] | eol <+ | .> [bl]

eos::= [bl] <; | eol>

eol::= carriage.return linefeed [formfeed]...

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 expression]

match::= bl string.primary bl pattern.expression

replacement::= bl variable bl pattern.expression equals [bl
        string.expression]

equals::= bl <= | ←>

variable::= \pattern.identifier identifier | & unprotected.keyword |
        string.variable

expression::= string.expression | \<string.primary [bl string.primary]...
        > pattern.expression

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

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

pattern.term::= pattern.primary [<bl . bl | bl $ bl> pattern.variable]...

pattern.primary::= pattern.identifier | pattern.primitive | @ pattern.va-
        riable | [*] string.primary | sum | ( [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.identifier identifier | literal | & <
        unprotected.keyword | protected.keyword> | string.variable |
        <? | ~ | = | +> string.primary | . variable | ( [bl]
        string.expression [bl] )

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

string.variable::=  $ string.primary | array.element | procedure.call

procedure.call::=  \pattern.primitive <identifier ( [bl] [parameter.list]
        [bl] )>

array.element::=  \pattern.identifier identifier <( | [> [bl]
        [parameter.list] [bl] <) | ]>

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

pc::=  [bl] , [bl]

identifier::=  letter [letter | digit | . | _]...

label::=  \<blank | tab | ; | + | = | . | *> char [\<blank | tab | ;>
        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 | 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

char::=  any.printing.character

protected.keyword::=  STFCOUNT | LASTNO | STNO | FNCLEVEL | STCOUNT |
        ERRTYPE | RTNTYPE | ALPHABET

unprotected.keyword::=  ABEND | ANCHOR | FULLSCAN | STNTRACE | MAXLNGTH
        | STLIMIT | ERRLIMIT | DENSITY | INPUT | OUTPUT | DUMP | STOPRUN

pattern.identifier::=  FAIL | FENCE | ABORT | ARB | BAL | SUCCEED | REM

pattern.primitive::=  <LEN | TAB | RTAB | POS | RPOS | SPAN | NSPAN |
        BREAK | BREAKX | ANY | NOTANY> ( [bl] <string.expression | *
        string.primary> [bl] ) | ARBNO( [bl] pattern.expression [bl] )
```

```
control.type ::= LIST | UNLIST | NOCODE | CODE | EJECT | SPACE [ bl
        integer.literal] | NOCROSS | CROSREF | FAIL | NOFAIL

declaration.type::=
        'OPTION' pc <'NO.STNO' | 'TIMER' | 'HASHSIZE=' integer.literal '> |
        'SNOBOL.MAIN' pc ' identifier ' |
        'SNOBOL.SUBPROGRAM' pc ' identifier ' |
        'PURGE.VARIABLE' pc <ALL | ' identifier.list '> |
        'UNPURGE.VARIABLE' pc ' identifier.list ' |
        'PURGE.LABEL' pc <ALL | ' label.list '> |
        'UNPURGE.LABEL' pc ' label.list ' |
        'PURGE.FUNCTION' pc <ALL | ' identifier.list '> |
        'UNPURGE.FUNCTION' pc ' identifier.list ' |
        'STRING' pc ' string.specifier.list ' |
        'INTEGER' pc ' identifier.list ' |
        'REAL' pc ' identifier.list ' |
        'RENAME' pc ' identifier ' pc ' identifier ' |
        'GLOBAL.VARIABLE' pc ' identifier.list ' |
        'GLOBAL.LABEL' pc ' label.list ' |
        'GLOBAL.FUNCTION' pc ' identifier.list ' |
        'EXTERNAL.VARIABLE' pc ' restricted.identifier.list ' |
        'ENTRY.VARIABLE' pc ' restricted.identifier.list ' |
        'EXTERNAL.LABEL' pc ' restricted.label.list ' |
        'ENTRY.LABEL' pc ' restricted.label.list ' |
        'EXTERNAL.FUNCTION' pc ' restricted.identifier.list ' |
        'ENTRY.FUNCTION' pc ' restricted.identifier ( [bl] [identifier.list]
                [bl] ) [[bl] identifier.list] ' [pc ' label '] |
        'EXTERNAL.FORTRAN.FUNCTION' pc ' fortran.identifier.list ' |
        'ENTRY.FORTRAN.FUNCTION' pc ' restricted.identifier ( [bl]
                [identifier.list] [bl] )' [pc ' label ']

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

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

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

string.specifier::= identifier ( integer.literal )

restricted.identifier.list::= restricted.identifier [pc restricted.iden-
        tifier]...

restricted.label.list::= restricted.identifier [bl restricted.identifier]...

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

fortran.identifier::= identifier [= <INTEGER | REAL>] ( integer.literal )

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

lnd::= letter | digit | _
```

APPENDIX 2

Predefined Symbols

1. GLOBAL and EXTERNAL variables
INPUT INPUTC OUTPUT OUTPUTC

2. GLOBAL and EXTERNAL labels
END FRETURN NRETURN RETURN

3. EXTERNAL.FORTRAN functions (all integer valued) AND(2) FREEZE(0) IEQ(2) IGE(0) IGT(2) ILE(2) ILT(2) INE(2) LSHIFT(2) NOT(1) OR(2) REMDR(2) RSHIFT(2) XOR(2)

4. Primitive pattern variables
ABORT ARB BAL FAIL FENCE REM SUCCEED

5. Primitive pattern functions
ANY ARBNO BREAK BREAKQ BREAKX LEN NOTANY NSPAN POS RPOS RTAB SPAN TAB

6. Predefined primitive functions
APPLY ARRAY CLOSE COLLECT CONVERT COPY DATA DATATYPE DATE DAYTIM DEFINE DETACH DIFFER DUPL EJECT ENTER EQ EXTIME GE GT IDENT INPUT INSERT INTEGER ITEM LE LGT LOOKUP LPAD LT NE OPEN OPSYN OUTPUT PROTOTYPE REAL RELEASE REPLACE REVERS RPAD SIZE SUBSTR TABLE TIME TRIM

APPENDIX 3

Runtime Errors

Conditionally Fatal

1. Illegal Data Type
2. Error in Arithmetic Operation
3. Erroneous Array or Table Reference
4. Null String in Illegal Context
5. Undefined Function or Operation
6. Erroneous Prototype
7. Dedicated String Overflow
8. Variable Not Present Where Required
9. Real to String Conversion Overflow
10. Illegal Argument to Primitive Function
11. Reading Error
12. Illegal I/O Unit
13. Limit on Defined Datatypes or Tables Exceeded
14. Negative Number in Illegal Context
15. String Overflow

Unconditionally Fatal

17. Error in FASBOL System
18. Return from Zero Level
19. Failure During Goto Evaluation
20. Insufficient Storage to Continue
21. Illegal Memory Reference
22. Limit on Statement Execution Exceeded

23. Object Exceeds Size Limit

24. Undefined or Erroneous Goto

25. [unused]

26. [unused]

27. Writing Error

28. Execution of Statement with Compilation Error

29. Failure Under NOFAIL

30. Divide Check

31. Arithmetic Overflow

REFERENCES

1. Santos, P. J., "FASBOL, A SNOBOL4 Compiler," Ph.D. Thesis, University of California, Berkeley, also Memorandum No. ERL-M314, Electronics Research Laboratory, University of California, Berkeley, December 1971.

2. Griswold, R. E., Poage, J. F., and Polonsky, I. P., The SNOBOL4 Programming Language, Prentice-Hall (1971) (Second Edition).

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