

AD-A099 498

COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE

F/G 9/2

SAM/SAL REPORT AND USER MANUAL.(U)

FEB 81 M A GALLUCCI

DAAG29-80-C-0094

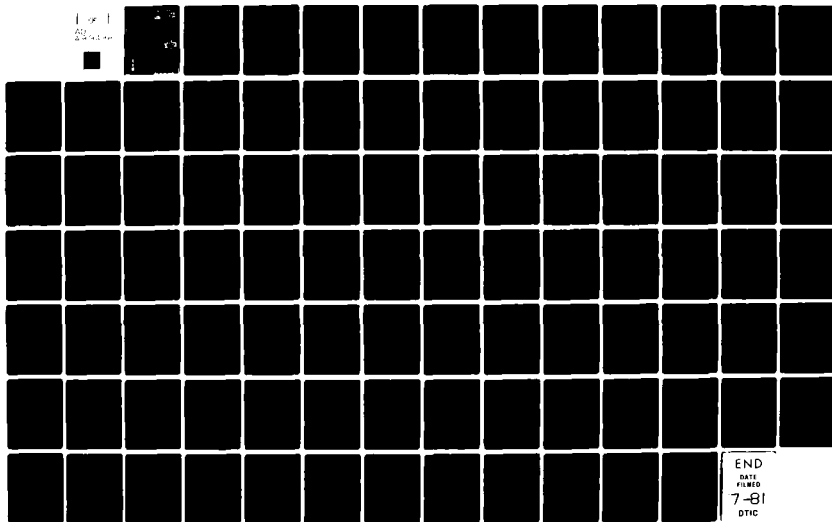
UNCLASSIFIED

CU-CS-198-81

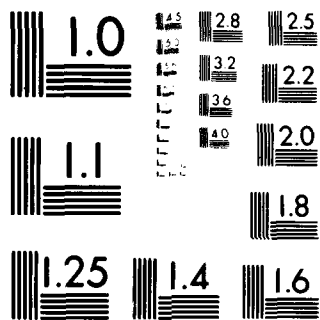
ARO-17124.4-M

ML

1 of 1
AD-A099 498



END
DATE
FILMED
7-81
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

AD A099 498

UNIVERSITY OF COLORADO



UNIVERSITY OF COLORADO

LEVEL II

12

SAM/SAL Report
and User Manual

Michael A. Gallucci

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-198-81

February, 1981

DTIC
ELECTE
JUN 1 1981
S E

This material is based upon work supported by the National Science Foundation grants #MCS8000017 and #MCS77-02194, Department of Energy grant #DE-AC02-80ER10718, and Army Research Office grant #DAAG 29-80-C-0094.

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) CU-CS-198-81	2. GOVT ACCESSION NO. AD-A099798	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) SAM/SAL SAL/SAM Report and User Manual	5. TYPE OF REPORT & PERIOD COVERED (9) Technical Rept.	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR (10) Michael A. Gallucci	8. CONTRACT OR GRANT NUMBER(s) (15) DAAG 29-80-C-0094 NSF-MCS 77-02194	9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. REPORT DATE (11) February 1981	11. NUMBER OF PAGES 85
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709	12. SECURITY CLASS. (of this report) Unclassified (12) 94	13a. DECLASSIFICATION/DOWNGRADING SCHEDULE NA
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (18) ARO	16. DISTRIBUTION STATEMENT (of this Report) (19) 17124.4-M Approved for public release; distribution unlimited.	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) NA		
18. SUPPLEMENTARY NOTES The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) attributed grammars, language syntax, language semantics, programming languages, annotated flowgraphs, static program analysis		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This document describes the SAM/SAL system implemented at the University of Colorado during 1980. SAM is a Static Analysis Machine with a Static Analysis Language, SAL. The main purpose of SAM/SAL is to specify arbitrary programming languages so that when programs in the specified language are run through the SAM/SAL system, an annotated flowgraph representation of the program is generated.		

THE FINDINGS IN THIS REPORT ARE NOT TO BE
CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE
ARMY POSITION, UNLESS SO DESIGNATED BY
OTHER AUTHORIZED DOCUMENTS.

Any opinions, findings, and conclusions or
recommendations expressed in this publica-
tion are those of the author and do not
necessarily reflect the views of the Na-
tional Science Foundation.

Accession For		
NTIS GRA&I	☒	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A		

Abstract

This document describes the SAM/SAL system implemented at the University of Colorado during 1980. SAM is a Static Analysis Machine with a Static Analysis Language, SAL. The main purpose of SAM/SAL is to specify arbitrary programming languages so that when programs in the specified language are run through the SAM/SAL system, an annotated flowgraph representation of the program is generated.

Table of Contents

1.	The SAM/SAL System	1
1.1	Introduction	1
1.1.1	Purpose	1
1.1.2	Motivation	3
1.2	Design Requirements	3
1.3	Syntax Notation	4
1.4	Language Outline	5
2.	Lexical Elements	6
2.1	Characters	6
2.2	Comments	6
2.3	Lexical Units	7
2.3.1	Names	7
2.3.2	Numbers	8
2.3.3	Literals	9
2.3.4	Delimiters	9
2.3.5	Lexical-Unit Restrictions	10
2.4	Spaces	10
2.5	Reserved Words	10
3.	Preamble	11
3.1	Scanner Specification	11
3.2	Data Structure Control	12
3.2.1	MAXSETS	12
3.2.2	MAXSETSIZE	13
3.2.3	MAXDPNODES	13
3.2.4	MAXEDGES	13
3.2.5	MAXPARSENODES	13
3.2.6	MAXSYM	13
3.2.7	MAXCHAR	13
3.2.8	MAXATTLCK	14
3.2.9	MAXPACKET	14
3.3	Grammar Output Control	14
4.	Declarations	16
4.1	Object Class Declarations	16
4.2	Actions	18
4.3	Flowgraph Node Types	18
4.4	Other Declarations	19
5.	Language Specifications	20
5.1	Grammar Attributes	20
5.1.1	Grammar Attribute Part : Syntax	20
5.1.2	Grammar Attribute Part : Semantics	21
5.1.2.1	Primitive Grammar Variables	21
5.1.2.2	Type Restrictions	22
5.2	Language Rules	23

5.2.1	Syntax Rules	23
5.2.1.1	Syntax Rule Syntax	24
5.2.1.2	Syntax Rule / Scanner Interface	24
5.2.1.3	Syntax Rule Restrictions	25
5.2.2	Semantic Rules	25
5.2.2.1	Evaluation Order of Semantic Rules	26
5.2.2.1.1	Interphase Ordering	26
5.2.2.1.2	Intraphase Ordering	27
5.2.2.1.3	Evaluation Order Restrictions	27
5.2.2.2	Semantic Rule Syntax	28
5.2.2.3	Semantic Rule Semantics	29
6.	Procedural Specifications	32
	References	34
A.	Using SAM/SAL on the CU CDC Cyber	35
A.1	Compiling a SAL Program, S	35
A.2	Generating the Evaluators	36
A.2.1	Parser Generation	36
A.2.2	Semantic Evaluator Generation	38
A.3	Using the Evaluators	38
A.3.1	Using the Parser	38
A.3.2	Using the Semantic Evaluator	39
A.4	Fancy Display	39
B.	The Standard Environment	40
B.1	Standard Types	40
B.1.1	Set Types	40
B.1.2	Symbol Types	40
B.1.3	Symbol Attribute Types	40
B.1.4	Object Class Types	40
B.1.5	Packet Types	41
B.1.6	Parameter Building Types	41
B.1.7	Callgraph Types	41
B.1.8	Parse-Tree Types	41
B.1.9	Other Types	41
B.2	Standard Procedures/Functions	41
B.2.1	Set Routines	41
B.2.2	Symbol Routines	44
B.2.3	Symbol Attribute Routines	46
B.2.4	Object-Class Routines	46
B.2.5	Packet Routines	47
B.2.5.1	Use-Table Routines	47
B.2.5.2	Flowgraph Routines	49
B.2.5.3	Expression-Tree Routines	51
B.2.6	Parameter Building Routines	52
B.2.7	Callgraph Routines	53
B.2.8	Parse-Tree Routines	55
B.2.9	Other Routines	55
C.	Output Tables Format	56
C.1	Data Structure Representations	56

Table of Contents

iii

C.2	File Format	59
D.	SAM/SAL System Sample Program	63
D.1	TURINGOL: A SAL Program	64
D.2	Sample TURINGOL Program	77
D.3	Output Report for Sample Program	78
D.4	Tables File for Sample Program	79
D.5	User-Readable Report of Tables File	82
D.6	Graphic Display of Tables File	85

Figures

Figure 1.1	SAM/SAL System	2
Figure C.1	Callgraph Node Structure	56
Figure C.2	Action Packet Structure	56
Figure C.3	Fp-Node Structure	56
Figure C.4	Flowgraph Node Structure	57
Figure C.5	Expression-Tree Node Structure	57
Figure C.6	Use-Table Node Structure	58

CHAPTER 1

The SAM/SAL System

1.1. Introduction

1.1.1. Purpose

This document describes the SAM/SAL system implemented at the University of Colorado during 1980. SAM, an acronym for Static Analysis Machine, is the current name given to the whole system. SAL, an acronym for Static Analysis Language, is the specification language provided by SAM through which most of SAM's descriptive capabilities are manifested. Since SAL is such an integral part of the overall system, the system will often be referred to as SAM/SAL.

The main purpose of SAM/SAL is to provide a capability for specifying arbitrary programming languages. A specification is to be aimed at generating annotated flowgraphs for programs written in the specified language. An annotation is regarded as a defined action occurring to a set of defined objects. As an example, in a specification of the language PASCAL, the user might want the PASCAL statement

$$X := Y+Z$$

to generate a single flowgraph node n which is annotated with the action REFERENCE to the set of objects $\{Y, Z\}$ and DEFINE to the set of objects $\{X\}$. In this case, the user must be able to declare X , Y and Z as objects of some class, declare the actions DEFINE and REFERENCE to be valid on subsets of objects from this class, and specify that the assignment statement above results in the creation of a flowgraph node.

Figure 1.1 gives a graphic description of how SAM/SAL works. At the top of the figure, a specification program S is submitted to SAL for compilation. Outputs from SAL are then fed to an automatic parser generator and to a semantic evaluator generator. A parser and semantic evaluator are then produced. A typical program U written in the language specified by S can then be fed to the generated parser; the parser output is in turn fed to the generated semantic evaluator; and the semantic evaluator in turn produces the desired annotated flowgraphs associated with the program U as specified by S .

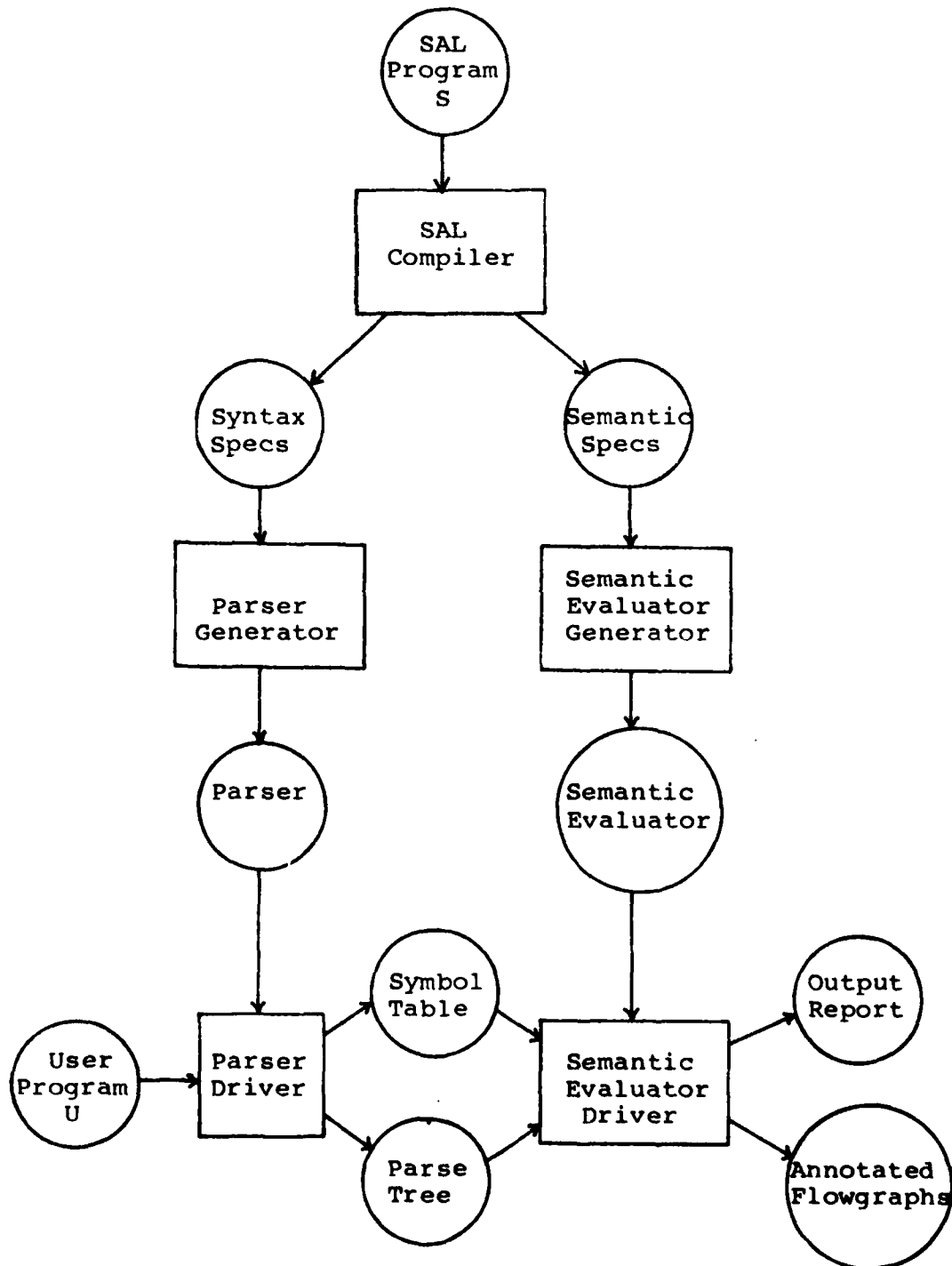


Figure 1.1 SAM/SAL System

Two output files are generated by the semantic evaluator.

(1) Listings File.

The listings file contains (a) any system error messages resulting from the semantic evaluation phase, (b) any special output requested by the user, and (c) a listing of program statistics.

(2) Tables File.

The tables file is automatically generated by the semantic evaluator upon successful semantic analysis of the input. Primarily, this file contains a dump of the symbol table, callgraph and flowgraphs generated by the semantic evaluator for the input.

Details on how this system works on the CU CDC-Cyber system are given in Appendix A.

1.1.2. Motivation

Research directed by Drs. Leon J. Osterweil and Lloyd D. Fosdick has lead to the design and implementation of a software tool, DAVE, which automatically detects certain static-semantic and data-flow errors in ANSI Standard Fortran programs [Fosd 76]. We recently completed a prototype of a revised version of DAVE which facilitates automatic modifications for some dialects of Fortran. Unfortunately, all semantic specifications must be manually redesigned for each such dialect.

The SAM/SAL system was motivated by the desire to have a fully automated system which eliminates the ad hoc manner of specifying programming languages and their dialects.

1.2. Design Requirements

SAL is a specification language in which other (procedural) programming languages are described. SAL was designed to have the power to capture all syntax and semantic descriptions of a large class of programming languages, specifically for the purpose of generating annotated flowgraphs for sample programs written in the specified language.

The device used for semantic specifications is a modified form of attributed grammars [Knuth 68]. It was the intent of this design to take advantage of existing reliable, portable software tools. At a high level, we were able to use an automatic parser generating system, CLEMSW, implemented on the CU CDC Cyber by Geoffrey Clemm [Clemm1

79]. The interface to this parse generator is automatically provided by the SAM/SAL system. The SAL compiler itself and the main driver for the semantic evaluator are written in a slightly extended version of PASCAL. This allows modifications and extensions to be made to the SAL compiler very easily, while providing reliable object code by taking advantage of an already existing compiler. (This also lends some portability to the SAM/SAL system -- a property not originally in the design requirements and not completely demonstrated yet). At the time of implementation, no CDC Cyber attribute grammar systems were known to be available. Consequently, the remainder of the SAM/SAL system was completely designed and implemented from scratch.

1.3. Synax Notation

Below is a description of the context-free syntax used to describe SAL. This notation is a variant of the Backus-Naur Form.

- (a) Angled brackets enclose grammar variables, for example

```
<PROGRAM>          <LIST OF ATTRIBUTES>
<STATEMENT>        <SUB 12>
```

- (b) Double-angled brackets enclose grammar variables whose syntax and semantic rules are given in the PASCAL Report and User's Manual [Jensen 74], for example

```
<<TYPE>>          <<PROCEDURE OR FUNCTION DECLARATION>>
<<IDENTIFIER>>
```

- (c) Reserved words and delimiters are enclosed in double quotes, for example

```
"::="           "; "           "BEGIN"           "OBJECT"
```

- (d) Square brackets enclose optional items, for example

```
<PROGRAM HEADING> [<PREAMBLE PART>] <DECLARATION PART>
```

- (e) Braces enclose a repeated item. The item may appear zero or more times, for example

```
<IDENT LIST> ::= <IDENTIFIER> {", " <IDENTIFIER>}
```

1.4. Language Outline

A SAL program is given by

```
<SAL PROGRAM> ::= "PROGRAM" <<IDENTIFIER>> ";"  
                  [<PREAMBLE SPECIFICATIONS>]  
                  <DECLARATION SPECIFICATIONS>  
                  <LANGUAGE SPECIFICATIONS>  
                  <PROCEDURE SPECIFICATIONS> "."
```

Each of the four specification parts are described in detail in Chapters 3 through 6 of this report. The program name <<IDENTIFIER>> has no functional purpose other than to name the SAL program.

A SAL program specifies a single programming language. In some cases, the SAL program itself may serve as the definition of the language. However, the intended use of a SAL program is only to capture enough of the semantics of a language (generally defined by other methods) to result in the generation of annotated flowgraphs for programs written in the specified language. As a result, not all language semantics are necessarily specified.

CHAPTER 2

Lexical Elements

This chapter defines the lexical elements of SAL.

2.1. Characters

The basic character set consists of letters, digits, special characters, the space character, and the end-of-line character (denoted by EOL).

(a) Letters

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

Implementation restrictions require that only upper case letters be allowed.

(b) Digits

0 1 2 3 4 5 6 7 8 9

(c) Special characters

" # \$ () [] < > + - / * , . ; : =

(d) The space character.

(e) The end-of-line character.

2.2. Comments

SAL recognizes two forms of comments:

(1) Inline comments

The construct

(* <any sequence of characters not containing "*" > *)

is an inline comment. Below are two examples of inline comments.

```
(* THIS IS A COMMENT ON ONE LINE *)
```

```
(*  
    THIS IS A COMMENT  
    OVER FOUR LINES  
)
```

(2) Endline comments

The constructs

```
# <any sequence of characters except EOL> EOL
```

or

```
$ <any sequence of characters except EOL> EOL
```

are endline comments. An example of an endline comment is

```
#  
# ENDLINE COMMENTS ARE  
# NICE FOR RUNNING TEXT  
# ALONG SIDE ACTUAL SAL  
# CODE.  
#
```

All endline comments begin with a "#" or "\$" and are terminated by the next end-of-line. An endline comment beginning with "\$" in addition causes a page-eject to occur starting with the next line following the EOL.

2.3. Lexical Units

The lexical units of SAL include names, numbers, delimiters, and literals. Except as explicitly provided, no lexical unit may contain imbedded spaces, comments, or EOL's.

2.3.1. Names

There are essentially three types of names recognized as primitive token units:

(a) Identifiers

The syntax for identifiers is as in the PASCAL Report, and as such its token unit type is denoted by <<IDENTIFIER>> (see Section 1.3(b)).

The length of an identifier is the number of characters comprising its string.

Examples

START SUB1 A2B3 SAL X12345

(b) Grammar identifiers

The syntax for <GRAMMAR IDENTIFIER> is

```
<GRAMMAR IDENTIFIER> ::=
    "<" <LETTER> {<LETTER>|<DIGIT>|" " } ">"
```

The length of a grammar identifier is the number of characters between its enclosing angled brackets.

Examples

```
<PROGRAM>                      <DECLARATION PART>
<STATEMENT LIST>              <FORTRAN 4>
```

(c) Qualified grammar identifiers

The syntax for the token unit <QUAL GRAMMAR IDENTIFIER> is

```
<QUAL GRAMMAR IDENTIFIER> ::=
    "<" <LETTER> {<LETTER>|<DIGIT>|" "}
    "(" <DIGIT> {<DIGIT>} ")" ">"
```

The length of a qualified grammar identifier is the number of characters between its enclosing angled brackets minus its parenthetical qualifier.

Examples

Qualified Grammar Identifier	Length
<PROGRAM(1)>	7
<FORTRAN 4(3)>	9
<STATEMENT LIST(2)>	14
<A1 23 B(1)>	7

2.3.2. Numbers

There are two types of numbers recognized by SAL as token units.

(a) Integers

The syntax for integers is

`<INTEGER> ::= <<UNSIGNED INTEGER>>`

Examples

12345 22222 5432 0

(b) Reals

The syntax for reals is

`<REAL> ::= <<UNSIGNED REAL>>`

Examples

1.4 25.6E-13 5.0E+12
0.3 1.498E1 3.14159

2.3.3. Literals

The token unit `<LITERAL>` is any sequence of characters not containing EOL and enclosed between two double quotes. To include a double quote in the literal, one writes the quote mark twice.

The length of a literal is the number of characters between the two enclosing double quotes. Two consecutive double quotes appearing within the literal are counted as a single character.

Examples

"1" and " " are two literals of length one.
"AB" and """"""" are two literals of length two.
"IS THIS A ""LITERAL"?" is a literal of length 20.

A literal must have a length greater than zero.

2.3.4. Delimiters

The characters

() [] + * / - , . ; : < > =

serve as one-character delimiters.

The character strings

<> <= >= := ..

serve as two-character delimiters.

The character string

::=

serves as a three-character delimiter.

2.3.5. Lexical-Unit Restrictions

The current SAM/SAL implementation restricts the length of an identifier (2.3.1(a)), grammar identifier (2.3.1(b)), and qualified grammar identifier (2.3.1(c)) to be no more than 30 characters. The length of a literal (2.3.3) may be no more than 15 characters.

2.4. Spaces

All lexical units may be separated by sequences of spaces, comments, or EOL's. The use of spaces, comments, and EOL's is mainly to provide readability and textual organization to the source program.

2.5. Reserved Words

The following identifiers are reserved words. The SAL programmer may not use reserved words in a context other than that explicit in the definition of SAL.

ACTION	DO	LABEL	PREAMBLE	SPECIFICATIONS
ACTIONS	DOWNT0	LANGUAGE	PROCEDURE	SYNTAX
AND	ELSE	MOD	PROGRAM	THEN
ARRAY	END	NIL	RECORD	TO
ATTRIBUTE	FILE	NODE	REPEAT	TOKEN
ATTRIBUTES	FLOWGRAPH	NOT	RETURN	TYPE
BEGIN	FOR	OBJECT	RULES	TYPES
CASE	FUNCTION	OF	SCANNER	UNTIL
CLASSES	GOTO	OR	SEMANTIC	VAR
CONST	GRAMMAR	OTHER	SEMANTICS	WHILE
DECLARATIONS	IF	PACKED	SET	WITH
DIV	IN			

CHAPTER 3

Preamble

A given implementation of SAM/SAL is expected to provide a standard environment of resources needed to aid the SAL programmer in a language specification. The standard environment should provide:

- (a) A default lexical scanner.
- (b) Predefined data-structures to represent
 - (1) Callgraph nodes and edges
 - (2) Flowgraph nodes and edges
 - (3) Object Classes
 - (4) Actions
- (c) Appropriate predefined accessing functions and procedures for these structures.

A SAL preamble is an optional specification which allows the user to extend or somewhat control this standard environment. Through the preamble, some implementation considerations (which are otherwise meant to be invisible to the user) are made visible. The form of a preamble is given by

```
<PREAMBLE SPECIFICATION> ::=  
    "PREAMBLE"  
    { <IMPLEMENTATION SPECS> }  
    "END" "PREAMBLE"
```

where the form and the content of the implementation specifications may vary from one installation to another. The current implementation specifications allowed on the CU CDC Cyber include (a) a capability to override the default lexical scanner by introducing another scanner more specific to the language being defined; (b) some capability to control data-structure memory allocation; and (c) the capability to control the form of the parser-grammar output. These three capabilities are elaborated below.

3.1. Scanner Specification

This would be given by

```
<IMPLEMENTATION SPEC> ::= <SCANNER>
```

where the form of the scanner is as described in [Clemm2 79]. The scanner must return four "kept" token types.

- (1) "IDNTFR" corresponding to <IDENTIFIER> in the user-specified grammar.
- (2) "STRING" corresponding to <STRING> in the user-specified grammar.
- (3) "CNSTNT" corresponding to <CONSTANT> in the user-specified grammar.
- (4) "FLOAT" corresponding to <FLOAT> in the user-specified grammar.

In addition,

- (5) if the user-specified grammar uses any special character as a literal token unit and that character always appears in literals of only length 1, then that character is to be returned as the token type "SINGLE" from the scanner, and
- (6) if the user-specified grammar uses any special character as a literal token unit and that character may appear in at least one literal of length greater than one, then that character is to be returned by the scanner as the token type "MANY".

3.2. Data Structure Control

All data structures provided by the standard environment have a default size. Most of these structures may have their default size changed by assigning a new size-value to an appropriate identifier in the preamble. Such assignments are given by the following syntax

```
<IMPLEMENTATION SPEC> ::=
    <PREAMBLE ID> "=" <INTEGER> ";"
```

where

```
<PREAMBLE ID> ::= "MAXSETS"      | "MAXSETSIZE" |
                  "MAXDPNODES"   | "MAXEDGES"  |
                  "MAXSYM"        | "MAXCHAR"   |
                  "MAXATTBLCK"    | "MAXPACKET"  |
                  "MAXPARSENODES"
```

3.2.1. MAXSETS (default 100)

This determines the number of SETS to be reserved in the SAM/SAL set-pool. The amount of memory allocated for

sets is then given by $\text{MAXSETS} * \text{MAXSETSIZE}$ words.

3.2.2. MAXSETSIZE (default 10)

This determines the number of words to be used in a set. For the CU CDC Cyber 59 bits of each word are used. Thus if $\text{MAXSETSIZE}=3$ then each set represents $3 \times 59 = 177$ objects.

3.2.3. MAXDPNODES (default 2000)

This determines the maximum number of dependency-graph nodes that will be reserved by the semantic evaluator phase. This graph controls the processing during semantic evaluation and is of no direct interest to the user except that its default size may be inadequate for semantic evaluation of large source programs written in the language specified.

3.2.4. MAXEDGES (default 2500)

This determines the maximum number of dependency-graph edges that will be reserved by the semantic evaluator phase. This may need to be explicitly set if the default value is inadequate for semantic evaluation of large source programs written in the specified language.

3.2.5. MAXPARSENODES (default 1000)

This determines the maximum number of parse-tree nodes that will be reserved for the semantic evaluator phase. This may need to be explicitly set if the default value is inadequate for semantic evaluation of large source programs written in the specified language. On the present implementation, each parse-tree node is two central memory words.

3.2.6. MAXSYM (default 250)

This determines the maximum number of symbol entries that will be reserved for the symbol table during the semantic evaluator phase. On the CDC Cyber, the total symbol table size can be given by

$$\text{MAXSYM} * (1 + \text{MAXCHAR}/10)$$

central memory words where MAXCHAR (a multiple of 10) is the maximum number of characters per symbol string.

3.2.7. MAXCHAR (default 10)

This determines the maximum number of characters per symbol string and should be a multiple of the number of characters which can be packed into a central memory word (in the case of the CDC Cyber series, a multiple of 10).

3.2.8. MAXATTBLCK (default 250)

This determines the maximum number of symbol attribute-blocks that will be reserved for the semantic evaluator phase. The attribute table size will then be given by $\text{MAXATTBLCK} * N$ words where N is the maximum number of symbol attributes declared for a given object class (see Section 4.1). Since a symbol may possess at most one attribute block, it is always sufficient for MAXATTBLCK to be less than or equal to MAXSYM .

3.2.9. MAXPACKET (default 250)

For the current implementation, a packet is a convenient storage unit which holds action annotations. As such, flowgraph nodes, expression-tree nodes, and use-table nodes are all packets. MAXPACKET determines the total number of packets to be reserved by the semantic evaluator phase. The amount of memory occupied by packet allocation is

$$\text{MAXPACKET} * (2 + \text{NUMACT})$$

words where NUMACT represents the total number of actions declared by the user (see Section 4.2).

3.3. Grammar Output Control

Often in practice a group of syntax rules are alternate rules for the same grammar variable. For example, the list of rules

```
<A> ::= <B>
<A> ::= <C>
<A> ::= <D>
```

can be expressed in an "alternatives" form

```
<A> ::= <B> | <C> | <D>
```

By default, since syntax rules in SAL can never explicitly be expressed in "alternatives" form (see Section 5.2.1.1), they are not listed to the grammar output file in this form. However, the parser generator used for the present SAL implementation will require significantly less memory if the grammar file generated by SAL were in "alternatives" form. This can be achieved by the `ALTERNATIVES` command in the preamble. The syntax for this command is

```
<IMPLEMENTATION SPEC> ::= "ALTERNATIVES" ";"
```

An example of a preamble is

PREAMBLE

```
MAXSYM = 500; # INCREASES DEFAULT SYMBOL TABLE SIZE.
MAXSETS = 800; # INCREASE DEFAULT SET-POOL SIZE.
#
ALTERNATIVES; # WRITE GRAMMAR OUTPUT FILE
# IN ALTERNATIVES FORM.
END PREAMBLE
```

CHAPTER 4

Declarations

Recall that the key idea (see Section 1.1) of a SAL program is to be able to specify actions on objects at flow-graph nodes. The primary purpose of the declarations section is to provide a mechanism for the user to declare classes of objects and actions for these objects in order to reflect the type of node annotations desired on the output flowgraphs. The syntax for the declarations specifications section is

```
<DECLARATION SPECIFICATION> ::=
    "DECLARATIONS"
        <OBJECT CLASS DECLARATIONS>
        <ACTION DECLARATIONS>
        [ <FLOWGRAPH NODE TYPES> ]
        <OTHER DECLARATIONS>
    "END" "DECLARATIONS"
```

<OBJECT CLASS DECLARATIONS>, <ACTION DECLARATIONS>, <FLOWGRAPH NODE TYPES>, and <OTHER DECLARATIONS> are elaborated further in Sections 4.1 through 4.4.

4.1. Object Class Declarations

It is convenient to think of objects as belonging to classes, each class having its own set of actions. In PASCAL, for example, the object classes might correspond to variables, labels, procedures, functions, and the main program. Of these five classes, the user may wish to associate one or more actions with only the "variables" class. In the object class declaration section, all object classes are declared, along with a (possibly empty) list of object attributes which objects in that class may possess. The syntax for the object class declarations is

```
<OBJECT CLASS DECLARATIONS> ::=
    "OBJECT" "CLASSES" ":"
        <OBJECT CLASS SPEC>
        [ <OBJECT CLASS SPEC> ]
```

```
<OBJECT CLASS SPEC> ::=
    <OBJECT CLASS> ":"
        "(" [ <OBJECT ATTRIBUTE LIST> ] ")" ";"
```

<OBJECT CLASS> ::= <<IDENTIFIER>>

<OBJECT ATTRIBUTE LIST> ::= <OBJECT ATTRIBUTE SPEC>
 { ";" <OBJECT ATTRIBUTE SPEC> }

<OBJECT ATTRIBUTE SPEC> ::=
 <OBJECT ATTRIBUTE> { "," <OBJECT ATTRIBUTE> }
 ":" <<TYPE>>

<OBJECT ATTRIBUTE> ::= <<IDENTIFIER>>

For example, in a specification of PASCAL one might have

```
OBJECT CLASSES :
  VARIABLES : ( );
  LABELS    : (FN:FGNODE);
  PROCEDURES: (PCALL:CALLPTR;PENTRY:FGNODE);
  FUNCTIONS : (FCALL:CALLPTR;FENTRY:FGNODE);
```

This declares four object classes: VARIABLES, LABELS, PROCEDURES, and FUNCTIONS. The class VARIABLES has no object attributes associated with it. The object class LABELS has a single attribute, FN, which is of the predefined flowgraph node descriptor type FGNODE (see Section B.1.5). The object classes PROCEDURES and FUNCTIONS each have two attributes associated with them; the first (PCALL and FCALL, respectively) is of the predefined callgraph node descriptor type CALLPTR (see Section B.1.7) and is to hold the callgraph node for any object in either of these classes; the second (PENTRY and FENTRY, respectively) is to hold the "entry" flowgraph node for any object in these classes.

An object can be inserted into a declared object class via a semantic rule (see Section 5.2.2) or via a SAL procedure or function invoked by a semantic rule (see Section 4.4). Similarly, an attribute for an object can be given a value via a semantic rule or via a procedure or function invoked by a semantic rule.

An Object Attribute is different from a Grammar Attribute (Section 5.1) and it is important that the user does not confuse these two concepts. An Object Attribute annotates the object (or symbol) which possesses it. It may be defined, referenced, and redefined by use of Attribute Table accessing functions (Section B.2.3). A Grammar Attribute, on the other hand, annotates a parse tree node (or equivalently, the grammar variable which names that node), and is subject to the rigorous rules of attributed grammar

evaluation [Knuth 68]. As such, a Grammar Attribute may be defined or referenced in a semantic rule (Section 5.2.2), but may never be redefined.

4.2. Actions

Actions are declared for object classes. Each action may affect only one object class, however an object class may own zero or more actions. The syntax for action declarations is

```
<ACTION DECLARATIONS> ::= "ACTIONS" ":"
                                <ACTION DEFINITION>
                                { <ACTION DEFINITION> }
```

```
<ACTION DEFINITION> ::=
    <ACTION> { "," <ACTION> } ":"
    "ON" <OBJECT CLASS> ";"
```

```
<ACTION> ::= <<IDENTIFIER>>
```

Continuing with our PASCAL specification example, an action declaration might be

```
ACTIONS :
    DEFINE, REFERENCE, UNDEFINE : ON VARIABLES;
    USED : ON LABELS;
```

Such a declaration would allow the user to later associate subsets of the object class VARIABLES with the actions DEFINE, UNDEFINE, and REFERENCE, and associate subsets of the object class LABELS with the action USED.

4.3. Flowgraph Node Types

The user is allowed to declare mnemonic names for the node types of the flowgraphs to aid in program readability. These names may then be used in a SAL statement which sets the type for a particular flowgraph node. These mnemonic names are automatically retained by the semantic evaluator phase for error reporting or user displays. The syntax for the flowgraph node type declaration is

```
<FLOWGRAPH NODE TYPES> ::=
    "FLOWGRAPH" "NODE" "TYPES" ":"
    <NODE NAME> { "," <NODE NAME> } ";"
```

```
<NODE NAME> ::= <<IDENTIFIER>>
```

An example of this declaration form for PASCAL is

FLOWGRAPH NODE TYPES :

ENTRY, EXIT, ASSIGNMENT, GOTOSTMT, PROCCALL,
EMPTYSTMT, IFTEST, CASETEST, WHILETEST,
REPEATTEST, FORINIT, FORTEST, FORINCR, FORUNDF;

Each node name must be no more than ten characters in length.

4.4. Other Declarations

The SAL user will often find it necessary to create other procedures and functions based on the primitive capabilities provided by the Standard Environment. The newly created procedures and functions are typically higher level routines which characterize functional properties of the language being specified. The definitions of such procedures and functions are elaborated in the declaration specifications section of the SAL program. Any constants, types, or global variables may also be declared in this section. The syntax for this is given by

```
<OTHER DECLARATIONS> ::=  
    <<CONSTANT DEFINITION PART>>  
    <<TYPE DEFINITION PART>>  
    <<VARIABLE DECLARATION PART>>  
    <<PROCEDURE AND FUNCTION DECLARATION PART>>
```

Note that any of the four parts above may be empty (as per usual PASCAL syntax). The motivation for providing this declaration form in SAL will become more apparent in Chapter 5 (specifically, see Sections 5.1.2.2 and 5.2.2.3(e)).

CHAPTER 5

Language Specifications

The language specifications section contains the primary information to specify a desired programming language. The syntax for this section is

```
<LANGUAGE SPECIFICATIONS> ::=  
    "LANGUAGE" "SPECIFICATIONS"  
    <GRAMMAR ATTRIBUTE PART>  
    <LANGUAGE RULES>  
    "END" "LANGUAGE" "SPECIFICATIONS"
```

<GRAMMAR ATTRIBUTE PART> and <LANGUAGE RULES> are further elaborated in Sections 5.1 and 5.2, respectively.

5.1. Grammar Attributes

This subsection allows the user to declare all of the grammar variables to be used in the language specification. For each such grammar variable a (possibly empty) list of grammar attributes is also declared. Each such attribute must be given a type.

5.1.1. Grammar Attribute Part : Syntax

The syntax for the grammar attribute part is

```
<GRAMMAR ATTRIBUTE PART> ::=  
    "GRAMMAR" "ATTRIBUTES"  
    <GRAMMAR VAR ATTLIST>  
    { <GRAMMAR VAR ATTLIST> }  
    "END" "GRAMMAR" "ATTRIBUTES"  
  
<GRAMMAR VAR ATTLIST> ::= <GRAMMAR IDENTIFIER> ":"  
    { <GRAMMAR ATTLIST> } ";"  
  
<GRAMMAR ATTLIST> ::= <GRAMMAR ATT DECL>  
    { ";" <GRAMMAR ATT DECL> }  
  
<GRAMMAR ATT DECL> ::=  
    <GRAMMAR ATTRIBUTE> { "," <GRAMMAR ATTRIBUTE> }  
    ":" <TYPE>
```

<GRAMMAR ATTRIBUTE> ::= <<IDENTIFIER>>

An example of a grammar var attlist is

```
<LABELLED STATEMENT> :  
  LABELVAL : SYMBOL;  
  START, FINISH : FGNODE;
```

5.1.2. Grammar Attribute Part : Semantics

A grammar var attlist serves to declare a grammar variable and its associated grammar attributes. Such a declaration allows the user to later reference or define the attributed variables (see Section 5.2.2.2) constructed from a grammar variable and any one of its grammar attributes. For a more complete discussion of the use and meaning of grammar attributes, see [Knuth 68].

5.1.2.1. Primitive Grammar Variables

Four predefined grammar variables belong to the set of terminal symbols of any user-specified grammar in SAL. These four grammar variables are called primitive grammar variables and include

<IDENTIFIER>	<CONSTANT>
<FLOAT>	<STRING>

These are the only four grammar variables allowed in the set of terminals for any user-specified grammar in SAL. As mentioned later in the Syntax Rule / Scanner Interface section (5.2.1.2), these terminal grammar variables name parse tree leaf nodes associated with "kept" tokens ([Clemm2 79]) in the source code of the parsed program being analyzed. A kept token has two pieces of information of use to the SAL programmer: (1) a symbol descriptor identifying the object being kept, and (2) the token number for the occurrence of the object in the source text. As such, for each of the four primitive grammar variables there exists two predefined grammar attributes; namely, VALUE of the standard type SYMBOL (Section B.1.2) and TOKEN of the standard type INTEGER (Section B.1.9). The VALUE and TOKEN attributes of any primitive grammar variable are automatically set by SAM/SAL to contain the symbol descriptor and token number, respectively, of the associated token in the source text.

The SAL user must observe the following rules regarding the declaration of primitive grammar variables.

- (a) Only the four primitive grammar variables mentioned above may possess grammar attributes named VALUE and TOKEN.

- (b) A primitive grammar variable may possess no grammar attributes other than VALUE and TOKEN.
- (c) A user wishing to use any of the four primitive grammar variables must still declare those grammar variables (along with any of the two special grammar attributes VALUE or TOKEN desired) according to the syntax rules of Section 5.1.1.

This discussion on the Grammar Attribute Part in general and the Primitive Grammar Variables in particular is now best illustrated by the following example:

```

GRAMMAR ATTRIBUTES      #
  <PROGRAM> : ;         # NO ATTRIBUTES
                        #
  <IDENTIFIER> :        #
    VALUE : SYMBOL;     # WILL ONLY USE "VALUE" ATTRIBUTE
                        # OF THIS PRIMITIVE GRAMMAR VAR.
  <STRING> :            #
    VALUE : SYMBOL;     # WILL USE BOTH PREDEFINED ATTRI-
    TOKEN : INTEGER;    # BUTES FOR THIS PRIM. GRAMMAR VAR.
                        #
  <CONSTANT> :          #
    VALUE : SYMBOL;     # THIS IS OK, SINCE PRIMITIVE.
    NUM   : INTEGER;    # INVALID... "NUM" IS NOT A VALID
                        # ATTRIBUTE FOR A PRIM. GRAMMAR VAR
  <STATEMENT> :         #
    START : FGNODE;     # OK, SINCE "STATEMENT" IS NOT PRIM.
    VALUE : SYMBOL;     # INVALID... NONPRIMITIVE GRAMMAR
                        # VAR MAY NOT HAVE ATTRIBUTE NAMED
                        # "VALUE".
END GRAMMAR ATTRIBUTES #

```

Note that this example contains two (documented) errors. Also, the attributed variables (see Section 5.2.2.2) <IDENTIFIER>.VALUE, <STRING>.VALUE, and <CONSTANT>.VALUE are predefined to be the symbol descriptors to the identifier, string, and constant, respectively, in the symbol table. The attributed variable <STRING>.TOKEN is predefined to be the token number for the occurrence of the string in the source text associated with this parse-tree terminal. The attributed variable <STATEMENT>.START is not predefined and must be explicitly defined by a semantic rule (see Section 5.2.2).

5.1.2.2. Type Restrictions

For the current implementation of SAL, attribute types must be either an INTEGER or subrange of INTEGER. If a grammar attribute is conceived to be of some structured type T (e.g. a PASCAL RECORD type), then the user should define the type T in the type definition part and declare a

variable V in the variable declaration part of the declaration specification section (see Section 4.4), so that V is some array of type T. V then acts as a pool of resources of type T, and an index into V then acts as a descriptor to an object of type T. Since such an index is a subrange of INTEGER, this descriptor is a valid grammar attribute. This is in fact how SETS, FLOWGRAPH NODES, CALLGRAPH NODES, etc. are provided by the current Standard Environment. For any such pool of structured objects declared by the user, the user should also carefully provide accessing functions and procedures to (1) allocate and deallocate an object in the pool, and (2) set or get fields within such an object.

5.2. Language Rules

The syntax for the language rules subsection is

```

<LANGUAGE RULES> ::= "RULES"
                    <RULE>
                    { <RULE> }
                    "END" "RULES"

<RULE> ::= <SYNTAX RULE>
          "SEMANTICS"
          [ "OBJECT" "SPECIFICATIONS"
            <SEMANTIC RULE LIST> ]
          [ "ATTRIBUTE" "SPECIFICATIONS"
            <SEMANTIC RULE LIST> ]
          [ "FLOWGRAPH" "SPECIFICATIONS"
            <SEMANTIC RULE LIST> ]
          [ "ACTION" "SPECIFICATIONS"
            <SEMANTIC RULE LIST> ]
          [ "OTHER" "SPECIFICATIONS"
            <SEMANTIC RULE LIST> ]
          "END"

<SEMANTIC RULE LIST> ::= <SEMANTIC RULE>
                        { ";" <SEMANTIC RULE> }

```

The syntax rule of any language rule is said to "govern" all semantic rules in any semantic rule list of that same language rule. <SEMANTIC RULE> is further elaborated in Section 5.2.2.2.

5.2.1. Syntax Rules

The collection of syntax rules, when combined, are to form a context-free accepting grammar and tree-building grammar for the specified language. If the language being specified is not context free (e.g. Fortran 66), then the user must carefully define a powerful lexical scanner in the

preamble (see Section 3.1) to resolve all context-sensitive features.

5.2.1.1. Syntax Rule Syntax

The syntax of a syntax rule is

```

<SYNTAX RULE> ::=
    <GRAMMAR VARIABLE> "==" <SYNTAX EXPRESSION>

<GRAMMAR VARIABLE> ::= <GRAMMAR IDENTIFIER> |
    <QUAL GRAMMAR IDENTIFIER>

<SYNTAX EXPRESSION> ::= <SYNTAX UNIT> {<SYNTAX UNIT>}

<SYNTAX UNIT> ::= <GRAMMAR VARIABLE> | <LITERAL>
  
```

Examples

```

<PROGRAM> ::= <HEADING> ";" <DECLARATIONS> ";" <BODY> "."
<STMT LST(1)> ::= <STATEMENT> ";" <STMT LST(2)>
  
```

The presence of a qualifier in a grammar variable has no effect on the syntax rule. Qualifiers are a semantic device only (see Section 5.2.2.3(b)). Thus, the two syntax rules below are grammatically indistinguishable:

```

<IDENT LIST(1)> ::= <IDENT LIST(2)> "," <IDENTIFIER>

<IDENT LIST> ::= <IDENT LIST> "," <IDENTIFIER>
  
```

The length of a grammar variable is the length of the grammar identifier (Section 2.3.1(b)) or qualified grammar identifier (Section 2.3.1(c)) which it derives.

5.2.1.2. Syntax Rule / Scanner Interface

In Section 3.1 it was mentioned that four special token types must be provided by the lexical scanner. These types correspond to the "kept" tokens ([Clemm2 79]) in a given source stream, and correspond with the four primitive grammar variables mentioned in Section 5.2.1.2. Explicitly, this correspondence is given by

```

"IDNTRF"  <-->  <IDENTIFIER>
"STRING"  <-->  <STRING>
"CNSTNT"  <-->  <CONSTANT>
"FLOAT"   <-->  <FLOAT>
  
```

This correspondence is automatically known to SAM/SAL. All

final details of the interface protocol are automatically handled by SAM/SAL.

5.2.1.3. Syntax Rule Restrictions

There are four restrictions to the collection of syntax rules. The first two restrictions have to do with general requirements of a context-free grammar. The last two restrictions are due to implementation requirements peculiar to the automatic parser generator.

- (1) There must exist exactly one grammar variable (called the start variable) which is the left side of at least one syntax rule and which appears on the right side of no syntax rule.
- (2) The primitive grammar variables (see Section 5.1.2.1) may not appear on the left side of any syntax rule.
- (3) The right side of a syntax rule may not be empty. Unfortunately, this may force a large increase in the number of syntax rules than might otherwise be possible if the empty production were permitted.
- (4) The right side of a syntax rule may have at most seven syntax units.

5.2.2. Semantic Rules

The semantic rules specified in SAL may be partitioned into five phases: OBJECT SPECIFICATIONS, ATTRIBUTE SPECIFICATIONS, FLOWGRAPH SPECIFICATIONS, ACTION SPECIFICATIONS, and OTHER SPECIFICATIONS. The use of these phases is a simple variation on a pure attributed grammar as defined in [Knuth 68], and is explained as follows. After some practice at using the pure nonprocedural attributed grammar device, it became clear that a specification program using such a device was intellectually more manageable if it was at least conceived of as a sequence of successive phases, where the run-time completion of a phase could be characterized as the completion of some conceptual user-level task. In a SAL program, the user's job is to create objects (update a symbol table), possibly decorate these objects (create object attributes in an attribute table), build flowgraphs, annotate flowgraph nodes with actions, and possibly perform other miscellaneous activities on these structures. The five phases mentioned above are intended to correspond to these five conceptual activities. The semantic rules within a phase are directed toward performing these corresponding activities. The concept of partitioning semantic rules into phases thereby allows a user to build or update global structures (symbol table, attribute table, flowgraph node table, edge lists, action packets, etc.) without having to

pass copies of these large structures up and down the parse tree.

This might be better realized with the following illustration. In order to create an object attribute in the attribute table the object must first exist as a symbol table entry. A semantic rule relying on either the object being in the symbol table or one of its attributes being in the attribute table must not execute until such table updates have been made. One expensive (but pure) method of signalling the semantic rule that the updates it requires have indeed been made is to propagate a set of grammar attributes up and down the parse tree to signal the completion of the symbol table update phase, and then propagate another set of grammar attributes up and down the parse tree to signal the end of the attribute table creation phase. The propagation of such grammar attributes is costly both in terms of memory (as many as two extra attributes needed per parse tree node per phase) and in terms of time (each attribute would have to be readied, scheduled, and computed).

With the semantic rule partitioning introduced in SAL, all of this effort of defining and propagating extra grammar attributes for end-of-phase signalling can be eliminated or greatly reduced.

5.2.2.1. Evaluation Order of Semantic Rules

The collection of all semantic rules specify a nonprocedural set of instructions. It is generally not clear from the source text ordering of these rules what their actual evaluation order might be.

5.2.2.1.1. Interphase Ordering

The phase-partitioning mentioned above (5.2.2) has the following interpretation: no semantic rule in a given semantic phase can execute until all semantic rules in any preceding phase have executed. This of course implies an ordering to the semantic phases. Explicitly, this ordering is

- (1) The OBJECT SPECIFICATIONS phase is first and therefore has no preceding phase. All semantic rules in this phase are therefore constrained by no rules from other phases. The intent of this phase is to contain (among other rules) those semantic rules which update the symbol table by creating objects (symbols).
- (2) The ATTRIBUTE SPECIFICATIONS phase is second. All semantic rules in this phase execute only after the semantic rules in the OBJECT SPECIFICATIONS phase have executed. The intent of this phase is to be able to

rely on the existence of a completed symbol table from the previous phase so that any associated symbol attributes may now be added to the attribute table.

- (3) The FLOWGRAPH SPECIFICATIONS phase is third. The intent of this phase is to build flowgraph nodes and edges relying on the existence of a completed symbol table and attribute table. A semantic rule in this phase may execute only after all semantic rules in the OBJECT SPECIFICATIONS and ATTRIBUTE SPECIFICATIONS phase have executed.
- (4) The ACTION SPECIFICATIONS phase is fourth. The intent of this phase is to annotate the nodes in the flowgraphs created by the previous (FLOWGRAPH SPECIFICATIONS) phase. A semantic rule in this phase may execute only after all semantic rules in the OBJECT SPECIFICATIONS, ATTRIBUTE SPECIFICATIONS, and FLOWGRAPH SPECIFICATIONS phases have executed.
- (5) The OTHER SPECIFICATIONS phase is fifth and last. The intent of this phase is to allow the specification of any additional semantic rules which may rely on the existence of all tables and structures completed by the previous four phases. A semantic rule in this phase may execute only after all semantic rules in any of the other four phases have executed.

5.2.2.1.2. Intraphase Ordering

Within a phase semantic rules are executed in an order determined by their dependencies on the other grammar attributes (see [Knuth 68]). In general this will not be a total order in that at any given moment more than one semantic rule may be ready for execution. The determination of grammar attribute dependencies, detection of which semantic rules at a given moment are ready for execution, and scheduling of all "ready" rules for execution is automatically handled by the SAM/SAL semantic evaluator.

An additional intraphase ordering imposed by SAL is that all assignment rules are executed before any procedure rule.

5.2.2.1.3. Evaluation Order Restrictions

It is possible to have a collection of semantic rules which cannot all execute. A simple example of this is illustrated by the following language rule.

```

<A> ::= <B>
  SEMANTICS
    OTHER SPECIFICATIONS
      <A>.ATT1 := F1(<B>.ATT1);
      <B>.ATT1 := F2(<A>.ATT1)
  END

```

From the first semantic rule in the example it is clear that <A>.ATT1 cannot be evaluated until after the evaluation of .ATT1. But from the second rule we see that .ATT1 cannot be evaluated until after the evaluation of <A>.ATT1. From the point of view of the SAM/SAL scheduler, a deadlock exists.

A SAL program in which all semantic rules can be evaluated without interdependency conflicts is called well defined. A valid SAL program must be well-defined, and the user must exercise care to ensure this behavior. The detection of any violations of a well-defined program occurs in the semantic evaluation phase and not during program compilation.

Research performed by other authors ([Boehm 76], [Jazay 75], [Kasten 78], and [Kenned 76]) has been done to investigate methods for improving semantic evaluation by enforcing a fixed evaluation strategy on the attribute grammar. In all cases, these improvements were achieved by restricting the class of attribute grammars accepted from the well-defined class above.

5.2.2.2. Semantic Rule Syntax

The syntax for a semantic rule is

```
<SEMANTIC RULE> ::= <ASSIGNMENT RULE> | <PROCEDURE RULE>
```

```
<ASSIGNMENT RULE> ::=
  <ATTRIBUTED VARIABLE> "==" <SEMANTIC EXPRESSION>
```

```
<PROCEDURE RULE> ::=
  <<IDENTIFIER>> {"(" <SEMANTIC EXPRESSION LIST> ")" }
```

```
<SEMANTIC EXPRESSION LIST> ::=
  <SEMANTIC EXPRESSION> {"," <SEMANTIC EXPRESSION>}
```

```
<ATTRIBUTED VARIABLE> ::=
  <GRAMMAR VARIABLE> "." <GRAMMAR ATTRIBUTE>
```

```

<SEMANTIC EXPRESSION> ::=
    <SEMANTIC SUBEXPRESSION>
    <SET OP> <SEMANTIC SUBEXPRESSION> |
    <SEMANTIC SUBEXPRESSION>

```

```

<SET OP> ::= "UNION" | "INTERSECTION" | "MINUS"

```

```

<SEMANTIC SUBEXPRESSION> ::=
    <SEMANTIC TERM> <ADD OP> <SEMANTIC TERM> |
    <SEMANTIC TERM>

```

```

<ADD OP> ::= "+" | "-"

```

```

<SEMANTIC TERM> ::=
    <SEMANTIC FACTOR> <MULT OP> <SEMANTIC FACTOR> |
    <SEMANTIC FACTOR>

```

```

<MULT OP> ::= "*" | "/"

```

```

<SEMANTIC FACTOR> ::=
    <INTEGER> | <REAL> | <<IDENTIFIER>> |
    <FUNCTION REFERENCE> | <ATTRIBUTED VARIABLE> |
    <<SIGN>> <SEMANTIC FACTOR> |
    "(" <SEMANTIC EXPRESSION> ")"

```

```

<FUNCTION REFERENCE> ::=
    <<IDENTIFIER>> { "(" <SEMANTIC EXPRESSION LIST> ")" }

```

5.2.2.3. Semantic Rule Semantics

- (a) For an attributed variable appearing in any semantic rule, the following must hold.
 - (1) The grammar attribute composing the attributed variable must appear in the grammar attribute list for the declaration of the associated (unqualified) grammar identifier (see Section 5.1.2).
 - (2) The grammar variable part of the attributed variable must appear in the syntax rule governing (see Section 5.2) the semantic rule containing the attributed variable.
- (b) A qualified grammar identifier is syntactically interpreted no differently than the same grammar identifier

without the qualifier. However, in semantic rules such qualifiers are often needed to distinguish between two or more occurrences of the same grammar variable. This is best illustrated by an example of a language rule:

```
<ID LIST(1)> ::= <ID> "," <ID LIST(2)>
  SEMANTICS
    OBJECT SPECIFICATIONS
      <ID>.ENVIRON := <ID LIST(1)>.ENVIRON
    END
```

The syntax rule describes a subtree of the parse tree rooted at <ID LIST(1)> and having two sons <ID> and <ID LIST(2)>. The semantic rule explicitly assigns to the ENVIRON attribute of <ID> the ENVIRON attribute of the root of this subtree. Without qualifiers on <ID LIST> such a semantic rule would be ambiguous since <ID LIST>.ENVIRON could refer to the ENVIRON attribute of either the subtree root or the second son.

- (c) The qualifier numbering within a language rule is up to the user. The only restriction is that a specific qualified grammar variable may appear at most once in a given syntax rule. Thus the following is invalid

```
<A(1)> ::= <A(1)> "ELSE" <B>
```

since the qualified grammar variable <A(1)> appears twice in the same syntax rule. The following two examples are valid

```
<A> ::= <A> "ELSE" <B>
```

```
<B(1)> ::= <B(2)> <A(1)> <A(2)>
```

In the first example, since <A> is not qualified it may appear more than once in the syntax rule. However it may not appear as part of an attributed variable in a semantic rule since such an attributed variable would result in an ambiguous reference. In the second example, each qualified grammar variable is correctly used at most once in the syntax rule.

- (d) The length of a grammar attribute is the number of characters in the attribute name. The length of an attributed variable is the length of its grammar variable part (Section 5.2.1.1) plus the length of its grammar attribute part plus one. For example, the grammar attribute ENVIRON has length 7, and the attributed variable <ID LIST(12)>.ENVIRON has length 15. The current SAM/SAL implementation restricts the length of any grammar attribute and attributed variable to be no more than 30.

- (e) A Procedure Rule or Function Reference may apply to any procedure or function declared either in the Standard Environment (see Appendix B) or in the Other Declarations section (see Section 4.4).

CHAPTER 6

Procedural Specifications

This final specification section of a SAL program allows the user to perform any post semantic computations to augment the output listing file of the semantic evaluator phase of SAM/SAL. Any computations in this section will automatically occur after all semantic rules have been computed, and after the symbol table, callgraph tables, and flowgraph tables have been dumped to the output tables file. Thus any computations occurring in this section cannot alter the output tables file. The computations may (and indeed are intended to) add to the output listing file. All global variables provided by the Standard Environment or declared by the user in the Other Declarations part (4.4) are available for use here. This specification section was added to the SAL language to provide the user with some post semantic control. It is expected that in most SAL programs there will no code in this specification section. The syntax for the procedural specifications section is

```
<PROCEDURAL PART> ::=
  "PROCEDURE" "SPECIFICATIONS"
    <OTHER DECLARATIONS>
    "BEGIN"
      <<STATEMENT>>
      { ";" <<STATEMENT>> }
    "END"
  "END" "PROCEDURE" "SPECIFICATIONS"
```

where <<STATEMENT>> has the usual PASCAL syntax and semantics, and in particular may be empty. Examples of a procedural specification are

```
(1)  PROCEDURE SPECIFICATIONS      # EXAMPLE OF AN EMPTY
      BEGIN                        # PROCEDURE SPECIFICATION
      END                          #
      END PROCEDURE SPECIFICATIONS #
```

and

(2) PROCEDURE SPECIFICATIONS

```
(*
  WRITE THE COMPLETED SYMBOL TABLE TO
  THE STANDARD FILE "OUTPUT".
*)
VAR
  SYM : SYMBOL;
BEGIN
  WRITELN("  DUMP OF SYMBOL TABLE");
  FOR SYM:=1 TO NUMSYM DO
    BEGIN
      WRITE("  ",SYM," ");
      WRITESYM(OUTPUT,SYM);
      WRITELN
    END
  END
END PROCEDURE SPECIFICATIONS
```

References

- [Bochm 76] Bochmann, Gregor V. "Semantic Evaluation from Left to Right", CACM, Vol. 19, No. 2, (Feb., 1976), pp. 55-62.
- [Clemm1 79] Clemm, G. M. "CLEMSW User's Manual", Tech. Rep. CU-CS-167-79, Dept. of Computer Science, Univ. of Colorado at Boulder, Boulder, Colo., November, 1979.
- [Clemm2 79] Clemm, G. M. "FSCAN Report and User's Manual", Tech. Rep. CU-CS-166-79, Dept. of Computer Science, Univ. of Colorado at Boulder, Boulder, Colo., November, 1979.
- [Fosd 76] Fosdick, L. D., and Osterweil L. J. "Data Flow Analysis In Software Reliability", Computing Surveys, Vol. 8, No. 3, (Sept., 1976), pp. 305-330.
- [Jazay 75] Jazayeri, M., and Walter, K. G. "Alternating Semantic Evaluator", Proceedings from ACM Annual Conference 1975, (Oct., 1975), pp. 230-234.
- [Jensen 74] Jensen, K., and Wirth N. PASCAL: User Manual and Report, 2nd ed., Springer-Verlag, New York (1974).
- [Kasten 78] Kastens, U. "Ordered Attribute Grammars", Technical Report, Institut fur Informatik II, Universitat Karlsruhe, Bericht Nr. 7/78.
- [Kenned 76] Kennedy, K., and Warren, S. K. "Automatic Generation of Efficient Evaluators for Attribute Grammars", Proceedings on 3rd Symposium on Principles 6), pp. 32-49.
- [Knuth 68] Knuth, D. E. "Semantics of Context-Free Languages", Mathematical Systems Theory, Vol. 2, No. 2, (June, 1968), pp. 127-145.

APPENDIX A

Using SAM/SAL on the CU CDC Cyber

Each of the phases listed below uses special files built for the SAM/SAL system. The names of these files, and the CU projects and CCID's under which they are accessed may change. The file names, projects, and CCID's given below are valid as of January, 1981.

A.1 Compiling a SAL Program, S

Nine output files are generated by the SAL compiler. Of these, six have SAM/SAL system names which are not to be altered by the user, and thus do not explicitly appear in the compile command. To compile a SAL source program, S:

```
GET, SAL=SALTRAN/PAPM, J973.  
SAL, S, SALIST, SCANNER, GRCLEM.
```

The single input file is:

S User specified source file to be compiled.

The nine output files are:

SALIST	Listing file. This includes a paginated copy of the original source text with line numbers, error diagnostics, cross-reference information, and program statistics.
SCANNER	Scanner file. This file contains the default or user-defined scanner specifications to be used by FSCAN.
GRCLEM	Grammar file. This file contains all syntax rules in the form expected by the tree-builder phase of parse generation.
DECLF	Declarations file. This file contains all declarations as specified in 4.4.
EVALF	Command Evaluation file. This file contains all semantic rules translated into PASCAL statements.
DGRAPHF	Dependency-Graph file. This file contains the sequence of PASCAL commands generated by SAL to build the dependency graph for any program in the

specified language for analysis by the semantic evaluator.

CNSTMOD Constants file. This file contains all constants which govern the size of the Standard Environment data structures.

PTCLF Productions Table file. This file contains PASCAL code for creation of the productions table in the semantic evaluator.

SALBODY Body file. This file contains PASCAL code as specified in Chapter 6.

A.2 Generating the Evaluators

A.2.1 Parser Generation

To automatically generate a parser for the language specified by S, the two output files SCANNER and GRCLEM from SAL are needed. Parse generation proceeds over several phases.

Phase 1. Process Scanner Specifications.

GET, FSCAN/PAPM, J973.
FSCAN, SCANNER, SCNLST, TBL1, ERRSCN.

The single input file is:

SCANNER Output from SAL compiler.

The three output files are:

SCNLST Scanner listing file.

TBL1 Fortran tables produced by FSCAN.

ERRSCN Error file.

If ERRSCN is empty, then you can proceed to the second phase of parse generation.

Phase 2. Process Scanner/Grammar Interface.

GET, SALTGB/PAPM, J973.
SALTGB, GRCLEM, TBL1, GRLIST, TBL2, TBL3, ERRRTGB.

The two input files are:

Appendix A

GRCLEM Output from SAL compiler.

TBL1 Output from FSCAN.

The four output files are:

GRLIST Tree-grammar listing file.

TBL2 Fortran tables produced by SALTGB.

TBL3 Grammar table produced by SALTGB.

ERRTGB Error file.

If ERRTGB is empty, you can proceed to the third phase of parse generation.

Phase 3. Create Fortran Grammar Tables.

GET, CLEMSW=CLMSWB/PAPM, J973.
CLEMSW, TBL3, CLMLIST, TBL4.

The single input file is:

TBL3 Output from SALTGB.

The two output files are:

CLMLIST CLEMSW listing file.

TBL4 Fortran table produced by CLEMSW.

If no errors were detected by CLEMSW, you can proceed to the final phase of parse generation.

Phase 4. Producing the Actual Parser.

To produce the actual parser for the language specified by S, the Fortran tables produced in the previous phases need to be compiled and edited into a parse-driver template. The KCL for this phase is:

REWIND, TBL1, TBL2, TBL4.
FTN, I=TBL1, L=0, R=BIN.
FTN, I=TBL2, L=0, B=BIN.
FTN, I=TBL4, L=0, B=BIN.
REWIND, BIN.
GET, PRSDRVB/PAPM, J973.
LIBEDIT, P=PRSDRVB, L=0, B=BIN, I=0, N=PARSE.

The three input files are:

TBL1 Output from FSCAN.
TBL2 Output from SALTGB.
TBL4 Output from CLEMSW.

The single output file is:

PARSE Object file for the parser for the language
 specified by S.

A.2.2 Semantic Evaluator Generation

To build a semantic evaluator for S:

GET, CENEVAL/PAPM, J973.
PASCAL, GENEVAL, GENLIST, SMEVAL.

The six (implicit) input files are:

EVALF, DGRAPHF, PTCLEF, SALBODY, DECLF, and CNSTMOD
 Output files from the SAL compiler.

The two output files are:

GENLIST PASCAL listing of the semantic evaluator.
SMEVAL Object file for the semantic evaluator for the
 language specified by S.

A.3 Using the Evaluators

This section describes how to use the parser and semantic evaluator created in Section A.2.

A.3.1 Using the Parser

To parse a program U in the language specified by S:

PARSE, U, ULIST, UTBL, UERR.

The two input files are:

PARSE The parser generated in A.2.1.
U A sample program in the language specified by S.

The three output files are:

ULIST A listing of file U with token numbers.

UTBL File containing symbol table and parse-tree for U.

UERR Listing of syntax errors in U.

A.3.2 Using the Semantic Evaluator

To perform the semantic evaluation of program U as specified by S:

```
GET,FTNSETB/PAPM,J973.  
LOAD,FTNSETB.  
SMEVAL,UTBL,SAMLIST,SAMTBL.
```

The two input files are:

SMEVAL The semantic evaluator generated in A.2.2.

UTBL The table-file generated by the parser for program U.

The two output files are:

SAMLIST Listing file containing (a) any system errors detected by the semantic evaluator, (b) any output requests issued by the user in S, and (c) program statistics for U.

SAMTBL This file contains the symbol table, call graph, and flowgraphs for program U.

A.4 Fancy Display

The current SAM/SAL system has a post phase which allows the user to get a readable listing of the tables file from the semantic evaluator. To invoke this display tool:

```
GET,SALPOST/PAPM,J973.  
SALPOST,SAMTBL,PLIST.
```

The single input file is:

SAMTBL Output from semantic evaluator.

The single output file is:

PLIST User-readable listing of input.

APPENDIX B

The Standard Environment

This appendix lists the Standard Environment TYPES, PROCEDURES and FUNCTIONS for the SAM/SAL system.

B.1 Standard Types

B.1.1 Set types

SETs	SET descriptor type
UNPSET	Unpacked representation of a set

B.1.2 Symbol Types

SYMBOL	Symbol descriptor type
OBJECT	Synonym for SYMBOL
SYMREP	Type for the character string of a symbol
SYMLNG	Subrange type for the length value of a symbol
UNPSTR	Unpacked type for SYMREP

B.1.3 Symbol Attribute Types

ATTRIBUTE	Attribute-name selector type
ATTBLOCK	Attribute-block descriptor

B.1.4 Object Class Types

OBJECTCLASS	Object-Class name type
-------------	------------------------

B.1.5 Packet Types

PACKET	Flowgraph node, expression-tree node, use-table node descriptor type
ACTION	Scalar type of user-defined actions
FGNODE	Synonym for PACKET
EXPNODE	Synonym for PACKET

B.1.6 Parameter Building Types

FPRMPTR	Formal parameter node descriptor
---------	----------------------------------

B.1.7 Callgraph Types

CALLPTR	Callgraph node descriptor type
---------	--------------------------------

B.1.8 Parse-Tree Types

PARSENODE	Parse-tree node descriptor type
-----------	---------------------------------

B.1.9 Other Types

These include all other primitive types provided by the PASCAL Report ([Jensen 74]). Specifically

INTEGER

REAL

CHAR

BOOLEAN

B.2 Standard Procedures/Functions

B.2.1 Set Routines

```
FUNCTION NEWSET:SETS;  
  (*  
    RETURNS A NEW (EMPTY) SET FROM THE SET-POOL.  
  *)
```

```
FUNCTION NULLSET:SETS;
```

```
  (*  
    SAME AS NEWSET  
  *)
```

```
PROCEDURE RETURNSET(VAR S:SETS);
```

```
  (*  
    RETURNS A SET TO THE SET-POOL.  
  *)
```

```
FUNCTION ISEMPY(S:SETS):BOOLEAN;
```

```
  (*  
    RETURNS TRUE <=> SET S IS EMPTY.  
  *)
```

```
PROCEDURE UNIONP(S1,S2:SETS; VAR RESULT:SETS);
```

```
  (*  
    RETURNS A NEW SET RESULT WHOSE VALUE IS THE  
    UNION OF SETS S1 AND S2.  
  *)
```

```
FUNCTION UNION(S1,S2:SETS):SETS;
```

```
  (*  
    SAME AS UNIONP EXCEPT THIS IS A FUNCTION, AND  
    HENCE HAS NO CONTROL OF GARBAGE COLLECTING ON  
    USED SETS.  
  *)
```

```
PROCEDURE INTERSECTP(S1,S2:SETS; VAR RESULT:SETS);
```

```
  (*  
    RETURNS A NEW SET RESULT WHOSE VALUE IS THE  
    INTERSECTION OF SETS S1 AND S2.  
  *)
```

```
FUNCTION INTERSECT(S1,S2:SETS):SETS;
```

```
  (*  
    SAME AS INTERSECTP EXCEPT THIS IS A FUNCTION,  
    AND HENCE HAS NO CONTROL OF GARBAGE COLLECTING  
    ON UNUSED SETS.  
  *)
```

```
PROCEDURE MINUSP(S1,S2:SETS; VAR RESULT:SETS);
  (*
    RETURNS A NEW SET RESULT WHOSE VALUE IS THE
    SET-DIFFERENCE OF SETS S1 AND S2.
  *)

FUNCTION MINUS(S1,S2:SETS):SETS;
  (*
    SAME AS MINUSP EXCEPT THIS IS A FUNCTION, AND
    HENCE HAS NO CONTROL OF GARBAGE COLLECTING ON
    UNUSED SETS.
  *)

PROCEDURE ASSIGNSET(S:SETS; VAR RESULT:SETS);
  (*
    RETURNS A NEW SET RESULT WHOSE VALUE IS SET
    S. (CREATES A COPY OF S).
  *)

PROCEDURE SETINSERT(ELEMENT:INTEGER; S:SETS);
  (*
    INSERTS ELEMENT INTO SET S.
  *)

FUNCTION ISMEMBER(ELEMENT:INTEGER; S:SETS):BOOLEAN;
  (*
    RETURNS TRUE <=> ELEMENT IS IN SET S.
  *)

FUNCTION ISSUBSET(S1,S2:SETS):BOOLEAN;
  (*
    RETURNS TRUE <=> S1 IS A SUBSET OF S2.
  *)

FUNCTION ISEQUAL(S1,S2:SETS):BOOLEAN;
  (*
    RETURNS TRUE <=> SET S1 AND SET S2 CON-
    TAIN THE SAME ELEMENTS.
  *)
```

```
PROCEDURE UNPACKSET(S:SETS; VAR UNP:UNPSET);
```

```
  (*
    UNPACK SET S INTO ARRAY UNP. THE ZEROETH
    ELEMENT OF UNP IS THE NUMBER, N, OF ELEMENTS
    IN S. THE NEXT N ELEMENTS IN UNP ARE THE
    ELEMENT VALUES OF S.
  *)
```

```
PROCEDURE WRITESET(VAR F:TEXT;S:SETS; IND:INTEGER;
                   VAR NUM:INTEGER);
```

```
  (*
    WRITE SET S TO FILE F, USING NO MORE THAN
    130 CHARACTERS PER LINE. START EACH NEW LINE
    WITH AN INDENTATION OF IND (IF IND>0) ELSE
    WITH AN INDENTATION OF 5. ALSO, IF IND=0
    THEN PRECEED FIRST LINE WITH NUMBER OF OBJECTS
    IN THE SET. RETURN THE NUMBER OF OBJECTS IN
    THE SET IN THE OUTPUT PARAMETER NUM.
  *)
```

B.2.2 Symbol Routines

```
PROCEDURE SETSYMMAX(SYM:SYMBOL; MAXATTR:INTEGER);
```

```
  (*
    SET MAXIMUM ATTRIBUTES ALLOWED BY SYM TO
    MAXATTR.
  *)
```

```
FUNCTION GETSYMMAX(SYM:SYMBOL):INTEGER;
```

```
  (*
    RETURNS MAXIMUM NUMBER OF ATTRIBUTES FOR SYMBOL
    SYM.
  *)
```

```
PROCEDURE SETSYMOBJ(SYM:SYMBOL; OBJC:OBJCTCLASS);
```

```
  (*
    SET OBJECT-CLASS FOR SYMBOL SYM TO BE OBJC.
  *)
```

```
FUNCTION GETSYMOBJ(SYM:SYMBOL):INTEGER;
```

```
  (*
    RETURN OBJECT-CLASS FOR SYMBOL SYM.
    (NOTE - RESULT IS INTEGER SINCE OBJCTCLASS
    CANNOT HAVE A VALUE OF ZERO.)
  *)
```

```
PROCEDURE SETSYMAUX(SYM:SYMBOL;AUX:INTEGER;VAL:INTEGER);
```

```
  (*  
    SET AUXILLARY ATTRIBUTE  AUX  OF SYMBOL  SYM  
    TO  VAL.  
  *)
```

```
FUNCTION GETSYMAUX(SYM:SYMBOL;AUX:INTEGER):INTEGER;
```

```
  (*  
    GET AUXILLARY ATTRIBUTE  AUX  OF SYMBOL  SYM.  
  *)
```

```
FUNCTION HASH(VAR STR:SYMREP; LEN:SYMLNG;  
              VAR WASTHERE:BOOLEAN):SYMBOL;
```

```
  (*  
    HASH STRING  STR  OF LENGTH  LEN.  RETURN SYMBOL  
    POINTER FOR STRING.  IF STRING ALREADY IN SYMBOL  
    TABLE RETURN ITS PREDEFINED HASHED VALUE AND  
    SET  WASTHERE  TO TRUE, OTHERWISE CREATE AND  
    RETURN A NEW SYMBOL POINTER AND SET  WASTHERE  
    TO FALSE.  
  *)
```

```
PROCEDURE GETSTRING(SYM:SYMBOL; VAR STR:SYMREP;  
                    VAR LEN:SYMLNG);
```

```
  (*  
    GET STRING VALUE <STR,LEN> ASSOCIATED WITH  
    SYMBOL  SYM.  
  *)
```

```
PROCEDURE WRITESYM(VAR F:TEXT; SYM:SYMBOL);
```

```
  (*  
    WRITE SYMBOL  SYM  TO FILE  F.  
  *)
```

```
PROCEDURE LISTSYMSET(VAR F:TEXT; S:SETS; INDENT:INTEGER);
```

```
  (*  
    WRITE THE SET OF SYMBOLS IN SET  S  TO FILE  F  
    USING NO MORE THAN 130 CHARACTERS PER LINE.  
    START EACH NEW LINE WITH AN INDENTATION OF  
    INDENT  SPACES.  
  *)
```

B.2.3 Symbol Attribute Routines

```
PROCEDURE SETSYMATT(SYM:SYMBOL; ATT:ATTBLOCK);
  (*
    SET ATTRIBUTE-FIELD OF SYMBOL SYM TO ATT.
  *)

FUNCTION GETSYMATT(SYM:SYMBOL):ATTBLOCK;
  (*
    GET ATTRIBUTE-FIELD OF SYMBOL SYM.
  *)

FUNCTION NEWATTBLCK: ATTBLOCK;
  (*
    RETURNS A NEW ATTRIBUTE-BLOCK POINTER AND UP-
    DATES TOTAL NUMBER OF ALLOCATED POINTERS.
  *)

PROCEDURE SETATT(ATT:ATTRIBUTE; SYM:SYMBOL; VAL:INTEGER);
  (*
    IF SYMBOL SYM HAS ACCES TO ATTRIBUTE ATT
    THEN SET ATTRIBUTE ATT OF SYM TO VAL, ELSE
    REPORT ERROR.
  *)

FUNCTION GETATT(ATT:ATTRIBUTE; SYM:SYMBOL):INTEGER;
  (*
    GET ATTRIBUTE ATT OF SYMBOL SYM PROVIDED
    SYM HAS ACCESS TO THIS ATTRIBUTE, ELSE REPORT
    ERROR.
  *)
```

B.2.4 Object-Class Routines

```
FUNCTION GETOBJSET(OBJCL:OBJCTCLASS):SETS;
  (*
    GET THE SET OF OBJECTS ASSOCIATED WITH OBJECT
    CLASS OBJCL.
  *)

PROCEDURE OBJCLDEBUG(VAR F:TEXT);
  (*
    DUMP ALL VALID OBJECT-CLASSES (INCLUDING THEIR
    OBJECTS) TO FILE F.
  *)
```

```
PROCEDURE SETMAXATT(OBJCL:OBJECTCLASS; VAL:INTEGER);
  (*
    SET MAXIMUM NUMBER OF ATTRIBUTES FOR OBJECT-
    CLASS OBJCL TO VAL.
  *)
```

```
FUNCTION GETMAXATT(OBJCL:OBJECTCLASS):INTEGER;
  (*
    RETURN MAXIMUM NUMBER OF ATTRIBUTES FOR
    OBJECT-CLASS OBJCL.
  *)
```

```
PROCEDURE INCLUDE(OBJ:OBJECT; OBJCL:OBJECTCLASS);
  (*
    INCLUDE OBJECT OBJ INTO OBJECT-CLASS OBJCL.
  *)
```

```
FUNCTION INCLASS(OBJ:OBJECT; OBJCL:OBJECTCLASS): BOOLEAN;
  (*
    RETURNS TRUE <=> OBJECT OBJ IS IN OBJECT-
    CLASS OBJCL.
  *)
```

B.2.5 Packet Routines

```
PROCEDURE SETACTION(ACTN:ACTION; P:PACKET; S:SETS);
  (*
    SET SPECIFIED ACTION OF PACKET P TO BE SET S.
  *)
```

```
FUNCTION GETACTION(ACTN:ACTION; P:PACKET):SETS;
  (*
    GET ACTION ACTN FROM PACKET P.
  *)
```

B.2.5.1 Use-Table Routines

```
PROCEDURE SETPCKPLST(P:PACKET; VAL:INTEGER);
  (*
    SET PARAM-LIST OF PACKET P TO VAL.
  *)
```

```
FUNCTION GETPCKPLST(P:PACKET):INTEGER;
  (*
    GET PARAM-LIST OF PACKET P.
  *)
```

```
PROCEDURE SETPCKNAME(P:PACKET; VAL:SYMBOL);
  (*
    SET REFERENCED SUBPROG OF PACKET P TO VAL.
  *)
```

```
FUNCTION GETPCKNAME(P:PACKET):SYMBOL;
  (*
    GET REFERENCED SUBPROG OF PACKET P.
  *)
```

```
PROCEDURE SETPCKEDGE(P:PACKET; VAL:PACKET);
  (*
    SET USE-EDGE OF PACKET P TO VAL.
  *)
```

```
FUNCTION GETPCKEDGE(P:PACKET):PACKET;
  (*
    GET USE-EDGE OF PACKET P.
  *)
```

```
PROCEDURE SETPCKNPRM(P:PACKET; VAL:INTEGER);
  (*
    SET NUMBER OF ACTUAL PARAMS OF PACKET P TO
    VAL.
  *)
```

```
FUNCTION GETPCKNPRM(P:PACKET):INTEGER;
  (*
    GET NUMBER OF ACTUAL PARAMS OF PACKET P.
  *)
```

```
PROCEDURE SETPCKREF(P:PACKET; VAL:INTEGER);
  (*
    SET CODE-REFERENCE OF PACKET P TO VAL.
  *)
```

```
FUNCTION GETPCKREF(P:PACKET):INTEGER;
  (*
    GET CODE-REFERENCE OF PACKET P.
  *)
```

```
FUNCTION NEWPACKET:PACKET;
  (*
    RETURN A NEW PACKET FROM THE PACKET-POOL.
  *)
```

B.2.5.2 Flowgraph Routines

FUNCTION NEWFGNNODE:FGNODE;

(*
 RETURN A NEW FLOWGRAPH NODE FROM PACKET POOL.
*)

PROCEDURE SETFGNTYP(F:FGNODE; VAL:INTEGER);

(*
 SET TYPE OF FLOWGRAPH NODE F TO VAL.
*)

FUNCTION GETFGNTYP(F:FGNODE):INTEGER;

(*
 GET TYPE OF FLOWGRAPH NODE F.
*)

PROCEDURE SETFGNEXP(F:FGNODE; E:PACKET);

(*
 SET EXPRESSION TREE ATTRIBUTE OF FLOWGRAPH NODE
 F TO E.
*)

FUNCTION GETFGNEXP(F:FGNODE):PACKET;

(*
 GET EXPRESSION TREE FOR FLOWGRAPH NODE F.
*)

PROCEDURE SETFGNNSON(F:FGNODE; VAL:INTEGER);

(*
 SET NUMBER OF SON-EDGES OF FLOWGRAPH NODE F
 TO VAL.
*)

FUNCTION GETFGNNSON(F:FGNODE):INTEGER;

(*
 GET NUMBER OF SON-EDGES FOR FLOWGRAPH NODE F.
*)

PROCEDURE SETFGNNPAR(F:FGNODE; VAL:INTEGER);

(*
 SET NUMBER OF PARENT-EDGES OF FLOWGRAPH NODE F
 TO VAL.
*)

```
FUNCTION GETFGNNPAR(F:FGNODE):INTEGER;
  (*
    GET NUMBER OF PARENT-EDGES FOR FLOWGRAPH NODE F.
  *)

PROCEDURE SETFGNISON(F:FGNODE; VAL:SETS);
  (*
    SET SON-EDGES OF FLOWGRAPH NODE F TO VAL.
  *)

FUNCTION GETFGNISON(F:FGNODE):SETS;
  (*
    GET SON-EDGES FOR FLOWGRAPH NODE F.
  *)

PROCEDURE SETFGNPAR(F:FGNODE; VAL:SETS);
  (*
    SET PARENT-EDGES OF FLOWGRAPH NODE F TO VAL.
  *)

FUNCTION GETFGNPAR(F:FGNODE):SETS;
  (*
    GET PARENT-EDGES FOR FLOWGRAPH NODE F.
  *)

PROCEDURE NNEDGE(FROMNODE, TONODE:FGNODE);
  (*
    CREATE A FLOWGRAPH EDGE FROM NODE FROMNODE
    TO NODE TONODE.
  *)

PROCEDURE NSEGE(FROMNODE:FGNODE; TOSET:SETS);
  (*
    CREATE A FLOWGRAPH EDGE FROM NODE FROMNODE
    TO EVERY NODE IN SET TOSET.
  *)

PROCEDURE SNEDGE(FROMSET:SETS; TONODE:FGNODE);
  (*
    CREATE A FLOWGRAPH EDGE FROM EVERY NODE IN SET
    FROMSET TO NODE TONODE.
  *)
```

```
PROCEDURE SSEDGE(FROMSET, TOSET: SETS);  
  (*  
    CREATE A FLOWGRAPH EDGE FROM EVERY NODE IN SET  
    FROMSET TO EVERY NODE IN SET TOSET.  
  *)
```

B.2.5.3 Expression-Tree Routines

```
FUNCTION NEWEXPNODE: EXPNODE;  
  (*  
    RETURN A NEW EXPRESSION-TREE NODE FROM THE  
    PACKET POOL.  
  *)
```

```
PROCEDURE SETEXPTYP(E: EXPNODE; TYP: INTEGER);  
  (*  
    SET TYPE OF EXPRESSION-TREE NODE E TO TYP.  
  *)
```

```
FUNCTION GETEXPTYP(E: EXPNODE): INTEGER;  
  (*  
    GET TYPE OF EXPRESSION-TREE NODE E.  
  *)
```

```
PROCEDURE SETEXPSON(I: INTEGER; E: EXPNODE; VAL: INTEGER);  
  (*  
    SET THE ITH SON OF EXPRESSION-TREE NODE E TO  
    VAL.  
  *)
```

```
FUNCTION GETEXPSON(I: INTEGER; E: EXPNODE): INTEGER;  
  (*  
    GET THE ITH SON OF EXPRESSION-TREE NODE E.  
  *)
```

```
PROCEDURE SETEXPUSE(E: EXPNODE; VAL: INTEGER);  
  (*  
    SET USE-LINK FIELD OF EXPRESSION-TREE NODE E  
    TO VAL.  
  *)
```

```
FUNCTION GETEXPUSE(E: EXPNODE): INTEGER;  
  (*  
    GET USE-LINK OF EXPRESSION-TREE NODE E.  
  *)
```

```
PROCEDURE SETEXPOBJ(E:EXPNODE; VAL:INTEGER);
  (*
    SET OBJECT FIELD OF EXPRESSION-TREE NODE E TO
    VAL.
  *)

FUNCTION GETEXPOBJ(E:EXPNODE):INTEGER;
  (*
    GET OBJECT OF EXPRESSION-TREE NODE E.
  *)
```

B.2.6 Parameter Building Routines

```
FUNCTION NEWFPNODE: FPRMPTR;
  (*
    RETURN NEW FORMAL-PARAMETER NODE.
  *)

PROCEDURE SETFPUSE(FP:FPRMPTR; L:INTEGER);
  (*
    SET USE-LINK OF PARAMETER NODE FP TO L.
  *)

FUNCTION GETFPUSE(FP:FPRMPTR):INTEGER;
  (*
    GET USE-LINK OF PARAMETER NODE FP.
  *)

PROCEDURE SETFPLINK(FP:FPRMPTR; L:INTEGER);
  (*
    SET FP-LINK OF PARAMETER NODE FP TO L.
  *)

FUNCTION GETFPLINK(FP:FPRMPTR):INTEGER;
  (*
    GET FP-LINK OF PARAMETER NODE FP.
  *)

PROCEDURE SETFPSET(FP:FPRMPTR; L:SETS);
  (*
    SET GLOBAL SET OF PARAMETER NODE FP TO L.
  *)
```

```
FUNCTION GETFPSET(FP:FPRMPTR):SETS;  
  (*  
    GET GLOBAL SET OF PARAMETER NODE  FP.  
  *)
```

B.2.7 Callgraph Routines

```
FUNCTION NEWCGRNODE:CALLPTR;  
  (*  
    RETURN A NEW CALLGRAPH POINTER  
  *)
```

```
PROCEDURE SETCGRNAME(C:CALLPTR; VAL:SYMBOL);  
  (*  
    SET NAME-FIELD OF CALLGRAPH NODE  C  TO  VAL.  
  *)
```

```
FUNCTION GETCGRNAME(C:CALLPTR):SYMBOL;  
  (*  
    GET NAME-FIELD OF CALLGRAPH NODE  C.  
  *)
```

```
PROCEDURE SETCGRNMFP(C:CALLPTR; VAL:INTEGER);  
  (*  
    SET NUM-PARAM-FIELD OF CALLGRAPH NODE  C  TO  
    VAL.  
  *)
```

```
FUNCTION GETCGRNMFP(C:CALLPTR):INTEGER;  
  (*  
    GET NUM-PARAM-FIELD OF CALLGRAPH NODE  C.  
  *)
```

```
PROCEDURE SETCGRFPL(C:CALLPTR; VAL:INTEGER);  
  (*  
    SET FP-FIELD OF CALLGRAPH NODE  C  TO  VAL.  
  *)
```

```
FUNCTION GETCGRFPL(C:CALLPTR):INTEGER;  
  (*  
    GET FP-FIELD OF CALLGRAPH NODE  C.  
  *)
```

```
PROCEDURE SETCGREDGE(C:CALLPTR; VAL:PACKET);
  (*
    SET EDGE-FIELD OF CALLGRAPH NODE C TO VAL.
  *)

FUNCTION GETCGREDGE(C:CALLPTR):PACKET;
  (*
    GET EDGE-FIELD OF CALLGRAPH NODE C.
  *)

PROCEDURE SETCGRNTRY(C:CALLPTR; VAL:PACKET);
  (*
    SET ENTRY-FIELD OF CALLGRAPH NODE C TO VAL.
  *)

FUNCTION GETCGRNTRY(C:CALLPTR):PACKET;
  (*
    GET ENTRY-FIELD OF CALLGRAPH NODE C.
  *)

PROCEDURE SETCGREXIT(C:CALLPTR; VAL:PACKET);
  (*
    SET EXIT-FIELD OF CALLGRAPH NODE C TO VAL.
  *)

FUNCTION GETCGREXIT(C:CALLPTR):PACKET;
  (*
    GET EXIT-FIELD OF CALLGRAPH NODE C.
  *)

PROCEDURE SETMAINCALL(VAL:CALLPTR);
  (*
    SET MAIN PROGRAM INDICATOR TO BE CALLGRAPH
    NODE C.
  *)

FUNCTION CALLEDGE(C:CALLPTR; OBJ:SYMBOL;
                  NUPARAM:INTEGER):PACKET;
  (*
    SEARCH USE-EDGES OF CALLGRAPH ENTRY C FOR A
    PACKET WITH NAME-FIELD OBJ, AND RETURN PACKET.
    IF PACKET DOESNT EXIST, THEN CREATE A PACKET
    WITH NAME OBJ.
  *)
```

B.2.8 Parse-Tree Routines

```
FUNCTION GETPRSATT(PND:PARSENODE;SEL:PRSSELECT):INTEGER;
```

```
  (*
```

```
    GET SELECTED FIELD FROM PARSE-TREE NODE #PND#.
```

```
  *)
```

```
PROCEDURE SETPRSATT(PND:PARSENODE; SEL:PRSSELECT;  
                   VAL:INTEGER);
```

```
  (*
```

```
    SETS THE SELECTED FIELD OF PARSE-TREE NODE  
    "PND" TO VAL.
```

```
  *)
```

```
PROCEDURE TREEDUMP(VAR F:TEXT);
```

```
  (*
```

```
    DUMP OF PARSE TREE IN TREE REPRESENTATION.
```

```
  *)
```

B.2.9 Other Routines

These include all the standard routines provided by the PASCAL Report ([Jensen 74, Appendix A]).

APPENDIX C

Output Tables Format

C.1 Data Structure Representations

CNAME	SYMBOL pointer to name of program-unit.
CEGE	PACKET pointer to first use-table node in edge-list.
CENTRY	PACKET pointer to ENTRY flowgraph node for this program-unit.
CNUMFP	Number of formal parameters for program-unit.
CFIRSTFP	SET pointer to first formal parameter node.

Figure C.1 Callgraph Node Structure

NOBJ	FOBJ	NOBJ is the number of objects in action-class , and FOBJ is a SET pointer for the first object.
NOBJ	FOBJ	
NOBJ	FOBJ	

Figure C.2 Action Packet Structure

MODE	Parameter-mode indicator (1 => "IN", 2 => "OUT", 3 => "IN/OUT").
LINK	

Figure C.3 Fp-Node Structure

PACKETS

FNODETYPE	Flowgraph node-type indicator.
FEXPTR	PACKET pointer to expression-tree root for this node.
FNSON	Number of sons for this node.
FFSON	PACKET pointer to first son.
FNPTR	Number of parents for this node.
FFPAR	PACKET pointer to first parent.
	} ACTIONS

Figure C.4 Flowgraph Node Structure

ENODETYPE	Expression-tree node-type indicator.
ESON1	PACKET pointer to first son.
ESON2	PACKET pointer to second son.
EUSE	PACKET pointer to link use-table information.
EOBJ	Symbol pointer to object whose action is currently unknown.
(unused)	} ACTIONS

Figure C.5 Expression-Tree Node Structure

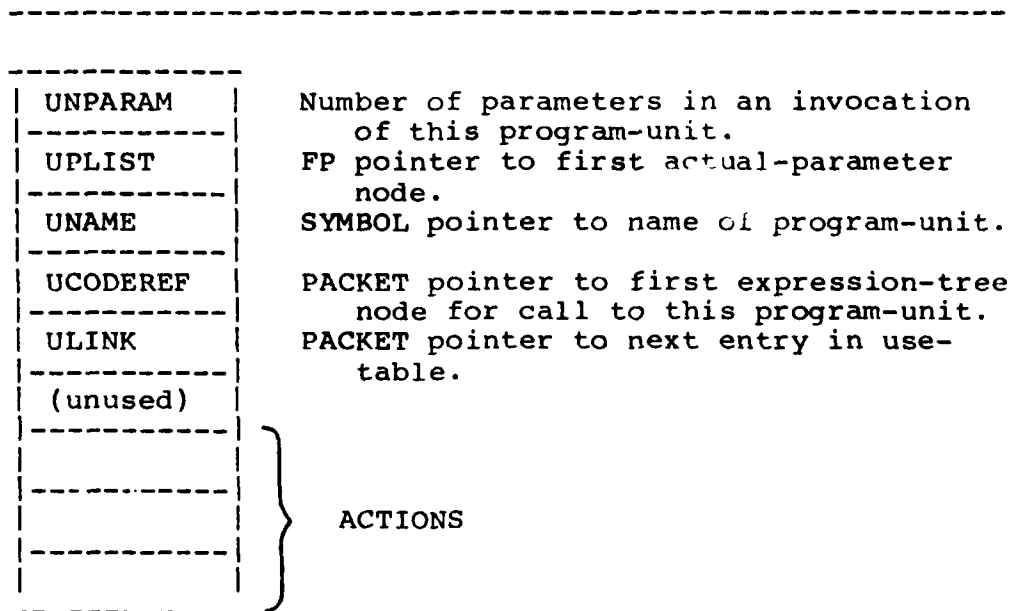


Figure C.6 Use-Table Node Structure

C.2 File Format

Pages 61 and 62 represent the format for the output Tables File. This file is a text file composed of lines (records). Each line holds no more than 130 characters. Below is a line-by-line description of the information on those pages.

Notice that lines 12, 17, 18, 19, 20, 22, 23, 24 and the ends of lines 13 and 14 describe sets or lists of objects. Such descriptions are represented by (1) the number, n , of objects in the list, and (2) a list of the n objects. The list may extend over a line boundary. After the last object in a given list is printed (within the file), the line holding that object is terminated.

- line 1 n_a is the number of user-defined actions.
- lines 2,3 Action names. Each string is exactly 10 characters long and each begins in column 2 of a new line.
- line 4 n_t is the number of expression-tree plus flowgraph node types.
- lines 5,6 Node-type names. Each string is exactly 10 characters long and each begins in column 2 of a new line.
- line 7 n_s is the number of Symbol Table entries (symbols).
- lines 8,9 Symbols. length_i is the number of characters in string $_i$. att_i is the Attribute Table descriptor owned by symbol $_i$. There is exactly one blank between att_i and string $_i$.
- line 10 Number of callgraph nodes, and callgraph node descriptor (index) for the main program.
- lines 11-32 Describe all information for callgraph node 1.
- line 11 Information for callgraph node 1. See Figure C.1.
- line 12 Information for callgraph node 1. See Figures C.1 and C.3.
- lines 13-15 Describe all use-table information for callgraph node 1.
- line 13 Information for use-table node indexed by $\text{UNODE}_{1,1}$ of callgraph node 1. See Figure C.6.
- line 14 Information for use-table node indexed by $\text{UNODE}_{1,n}$ of callgraph node 1. See Figure C.6.
- NOTE: The value for ULINK_i is UNODE_{i+1} .
- line 15 Zero. Indicates end-of-use-table information for callgraph node 1.
- lines 16-25 Describe all flowgraph information for flowgraph owned by callgraph node 1.

- line 16 Information for flowgraph node indexed by $FNODE_{1,1}$ of callgraph node 1. See Figure C.4.
- line 17 Set of sons for flowgraph node $FNODE_{1,1}$ of callgraph node 1.
- line 18 Set of parents for flowgraph node $FNODE_{1,1}$ of callgraph node 1.
- line 19 Set of objects in $Action-Class_1$ for flowgraph node $FNODE_{1,1}$ of callgraph node 1.
- line 20 Set of objects in $Action-Class_{n_a}$ (n_a given in line 1) for flowgraph node $FNODE_{1,1}$ of callgraph node 1.
- lines 21-24 Same as lines 16-20 except for flowgraph node indexed by $FNODE_{1,m}$. The total number of flowgraph nodes, m , is indeterminate.
- line 25 Zero. Indicates end-of-flowgraph information for callgraph node 1.
- lines 26-32 Describe all expression-tree information for callgraph node 1.
- line 26 Information for expression-tree node indexed by $ENODE_{1,1}$ of callgraph node 1. See Figure C.5.
- lines 27,28 Action sets (format as in lines 19, 20) for expression-tree node $ENODE_{1,1}$ of callgraph node 1.
- lines 29-31 Same as lines 26-28 except for expression-tree node $ENODE_{1,k}$ of callgraph node 1. The total number of expression-tree nodes, k , is indeterminate.
- line 32 Zero. Indicates end-of-expression-tree information for callgraph node 1.
- lines 33-34 Represent same information as lines 11-32 except for callgraph node 2.
- lines 35-36 Represent same information as lines 11-32 except for callgraph node $numcall$, where $numcall$ is given in line 10.

1. $\underline{n}_a$
2. $\underline{string}_1$
- $\vdots$
3. $\underline{string}_{n_a}$
4. $\underline{n}_t$
5. $\underline{string}_1$
- $\vdots$
6. $\underline{string}_{n_t}$
7. $\underline{n}_s$
8. $\underline{length}_1 \quad \underline{att}_1 \quad \underline{string}_1$
- $\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots$
9. $\underline{length}_{n_s} \quad \underline{att}_{n_s} \quad \underline{string}_{n_s}$
10. $\underline{numcall} \quad \underline{maincall}$
11. $\underline{CNAME}_1 \quad \underline{CEDGE}_1 \quad \underline{CENTRY}_1 \quad \underline{CEXIT}_1$
12. $\underline{CNUMFP}_1 \quad \underline{MODE}_{1,1} \quad \underline{MODE}_{1,2} \quad \dots \quad \underline{MODE}_{1,CNUMFP_1}$
13. $\underline{UNODE}_{1,1} \quad \underline{UNAME}_{1,1} \quad \underline{UCODEREF}_{1,1} \quad \underline{UNPARAM}_{1,1} \quad \underline{P}_{1,1,1} \quad \dots \quad \underline{P}_{1,1,UNPARAM_{1,1}}$
- $\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots$
14. $\underline{UNODE}_{1,n} \quad \underline{UNAME}_{1,n} \quad \underline{UCODEREF}_{1,n} \quad \underline{UNPARAM}_{1,n} \quad \underline{P}_{1,n,1} \quad \dots \quad \underline{P}_{1,n,UNPARAM_{1,n}}$
15. 0
16. $\underline{FNODE}_{1,1} \quad \underline{FNODETYPE}_{1,1} \quad \underline{FEXPTR}_{1,1}$
17. $\underline{FNSON}_{1,1} \quad \underline{SON}_{1,1,1} \quad \dots \quad \underline{SON}_{1,1,FNSON_{1,1}}$
18. $\underline{FNPAR}_{1,1} \quad \underline{PAR}_{1,1,1} \quad \dots \quad \underline{PAR}_{1,1,FNPAR_{1,1}}$
19. $\underline{NOBJ}_{1,1,1} \quad \underline{OBJ}_{1,1,1,1} \quad \dots \quad \underline{OBJ}_{1,1,1,NOBJ_{1,1,1}}$
- $\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots$
20. $\underline{NOBJ}_{1,1,n_a} \quad \underline{OBJ}_{1,1,n_a,1} \quad \dots \quad \underline{OBJ}_{1,1,n_a,NOBJ_{1,1,n_a}}$
- $\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots$
21. $\underline{FNODE}_{1,m} \quad \underline{FNODETYPE}_{1,m} \quad \underline{FEXPTR}_{1,m}$
22. $\underline{FNSON}_{1,m} \quad \underline{SON}_{1,m,1} \quad \dots \quad \underline{SON}_{1,m,FNSON_{1,m}}$
- $\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots$
23. $\underline{NOBJ}_{1,m,1} \quad \underline{OBJ}_{1,m,1,1} \quad \dots \quad \underline{OBJ}_{1,m,1,NOBJ_{1,m,1}}$
- $\vdots \quad \quad \quad \vdots \quad \quad \quad \vdots$
24. $\underline{NOBJ}_{1,m,n_a} \quad \underline{OBJ}_{1,m,n_a,1} \quad \dots \quad \underline{OBJ}_{1,m,n_a,NOBJ_{1,m,n_a}}$
25. 0

26. $\underline{\text{ENODE}}_{1,1}$ $\underline{\text{ENODETYPE}}_{1,1}$ $\underline{\text{ESON1}}_{1,1}$ $\underline{\text{ESON2}}_{1,1}$ $\underline{\text{EUSE}}_{1,1}$ $\underline{\text{EOBJ}}_{1,1}$

27. $\underline{\text{NOBJ}}_{1,1,1}$ $\underline{\text{OBJ}}_{1,1,1,1}$... $\underline{\text{OBJ}}_{1,1,1,\text{NOBJ}_{1,1,1}}$

⋮

⋮

⋮

28. $\underline{\text{NOBJ}}_{1,1,n_a}$ $\underline{\text{OBJ}}_{1,1,n_a,1}$... $\underline{\text{OBJ}}_{1,1,n_a,\text{NOBJ}_{1,1,n_a}}$

⋮

⋮

⋮

29. $\underline{\text{ENODE}}_{1,k}$ $\underline{\text{ENODETYPE}}_{1,k}$ $\underline{\text{ESON1}}_{1,k}$ $\underline{\text{ESON2}}_{1,k}$ $\underline{\text{EUSE}}_{1,k}$ $\underline{\text{EOBJ}}_{1,k}$

30. $\underline{\text{NOBJ}}_{1,k,1}$ $\underline{\text{OBJ}}_{1,k,1,1}$... $\underline{\text{OBJ}}_{1,k,1,\text{NOBJ}_{1,k,1}}$

⋮

⋮

⋮

31. $\underline{\text{NOBJ}}_{1,k,n_a}$ $\underline{\text{OBJ}}_{1,k,n_a,1}$... $\underline{\text{OBJ}}_{1,k,n_a,\text{NOBJ}_{1,k,n_a}}$

32. 0

33. $\underline{\text{CNAME}}_2$ $\underline{\text{CEDGE}}_2$ $\underline{\text{CENTRY}}_2$ $\underline{\text{CEXIT}}_2$

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

34. 0

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

35. $\underline{\text{CNAME}}_{\text{numcall}}$ $\underline{\text{CEDGE}}_{\text{numcall}}$ $\underline{\text{CENTRY}}_{\text{numcall}}$ $\underline{\text{CEXIT}}_{\text{numcall}}$

⋮

⋮

⋮

⋮

⋮

⋮

⋮

⋮

36. 0

APPENDIX D

SAM/SAL System Sample Program

This appendix presents an example of the various inputs to and outputs from the SAM/SAL system. The language specified by this example is TURINGOL, a simple language described in [Knuth 68].

Section D.1 lists a SAL program specifying TURINGOL. This listing corresponds to "SAL Program S" in Figure 1.1.

Section D.2 lists a sample TURINGOL program. This program corresponds to the "User Program U" in Figure 1.1.

Section D.3 lists the "Output Report" in Figure 1.1 for the sample program of Section D.2.

Section D.4 lists the "Annotated Flowgraphs" file in Figure 1.1 for the sample program of Section D.2. This file is the Tables File whose general format is given in Section C.2.

Section D.5 presents a user-readable listing of the Tables File of Section D.4.

Section D.6 gives a graphic illustration of the output for the sample program of Section D.2.

Note that TURINGOL allows no procedures or functions. As a result, (1) the output Tables File will always consist of a callgraph having only a single node; (2) the use-table list (dependent sons of a callgraph node) will be empty; and (3) no expression trees are necessary. These properties are apparent from the listing in Section D.5.

D.1 TURINGOL: A SAL Program

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11 07.31.

PAGE 1

```

1 0 PROGRAM TURINGOL;
2 0 *
3 0 * TURINGOL IS A SAMPLE LANGUAGE SPECIFIED IN:
4 0 *
5 0 *   KNUTH, D. E., "SEMANTICS OF CONTEXT-FREE LANGUAGES", MATHEMATICAL
6 0 *   SYSTEMS THEORY, VOL. 2, NO. 2, (JUNE 1968), PP. 127-145.
7 0 *
8 0 *   INSTEAD OF TURING-MACHINE DELTA FUNCTIONS, THIS SPECIFICATION
9 0 *   PRODUCES AN ANNOTATED FLOWGRAPH.
10 0 *
11 0 *   SINCE THIS IMPLEMENTATION CANNOT HANDLE THE EMPTY PRODUCTION,
12 0 *   THE EMPTY STATEMENT WILL BE DENOTED BY "NULL".
13 0 *
14 0
15 0 PREAMBLE
16 0
17 0   SCANNER SCANTOK:
18 0   SCANTOK -> (SPACES / TOKEN)* SPACES;
19 0   SPACES -> (' ' / 'EOL')* ;
20 0   SCANNER TOKEN:
21 0   TOKEN -> IDNT / '*' STRNG '*' / LITERAL / MISC1 / MISC2 ;
22 0   END TOKEN;
23 0   IDNT -> CHAR+ => "IDNTFR" ;
24 0   STRNG -> CHAR+ => "STRING" ;
25 0   LITERAL -> "." / "," / ";" / ":" / "[" / "]" => "SINGLE" ;
26 0   CHAR -> "A" / "B" / "C" / "D" / "E" / "F" /
27 0   "G" / "H" / "I" / "J" / "K" / "L" /
28 0   "M" / "N" / "O" / "P" / "Q" / "R" /
29 0   "S" / "T" / "U" / "V" / "W" / "X" /
30 0   "Y" / "Z" ;
31 0   MISC1 -> ".*" => "CNSTNT";
32 0   MISC2 -> ".*" => "FLOAT";
33 0   END SCANTOK.
34 0
35 0 END PREAMBLE

```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.31.

PAGE 2

```
36 0
37 0
38 0
39 0
40 0
41 0
42 0
43 0
44 0
45 0
46 0
47 0
48 0
49 0
50 0
51 0
```

DECLARATIONS

OBJECT CLASSES :

ALPHABET : ();

LABELS : (FN : FGNODE);

ACTIONS :

DECLARE, REF : ON ALPHABET;

USE, DEF : ON LABELS;

FLOWGRAPH NODE TYPES :

ENTRYSTMT, EXITSTMT, IFSTMT, EMPTYSTMT,

GOTOSTMT, MOVESTMT, PRINTSTMT;

\$ PROCEDURES AND FUNCTIONS

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.31.

PAGE 3

```

52 0
53 0 FUNCTION SETLABEL(L:OBJECT; S:FGNODE):INTEGER;
54 0 (*
55 0     SET FLOWGRAPH NODE FOR LABEL L TO BE S.
56 0 *)
57 0 BEGIN (* SETLABEL *)
58 1     SETATT(FN, L, S);
59 1     SETLABEL:=1
60 1 END (* SETLABEL *);
61 0
62 0 FUNCTION GETLABEL(L:OBJECT; CONTROL:INTEGER):FGNODE;
63 0 (*
64 0     GET FLOWGRAPH NODE ASSOCIATED WITH LABEL L.
65 0 *)
66 0 BEGIN (* GETLABEL *)
67 1     GETLABEL:=GETATT(FN, L)
68 1 END (* GETLABEL *);
69 0
70 0 PROCEDURE CHECKVAR(OBJ:OBJECT; TOKEN:INTEGER);
71 0 (*
72 0     REPORT SEMANTIC ERROR IF OBJ HAS NOT BEEN DECLARED
73 0     TO BE IN THE TAPE ALPHABET.
74 0 *)
75 0 BEGIN (* CHECKVAR *)
76 1     IF NOT INCLASS(OBJ,ALPHABET) THEN
77 1         BEGIN (* THEN *)
78 2             WRITELN;
79 2             WRITELN(" ":10,"SEMANTIC ERROR:");
80 2             WRITE(" ":20,"SYMBOL ");
81 2             WRITESYM(OUTPUT,OBJ);
82 2             WRITELN(" AT TOKEN ",TOKEN:1," NOT DECLARED.")
83 2         END (* THEN *)
84 1     END (* CHECKVAR *);
85 0
86 0 PROCEDURE CHECKLABEL(OBJ:OBJECT; TOKEN:INTEGER);
87 0 (*
88 0     REPORT SEMANTIC ERROR IF OBJ HAS NOT BEEN DEFINED
89 0     AS A LABEL.
90 0 *)
91 0 BEGIN (* CHECKLABEL *)
92 1     IF NOT INCLASS(OBJ,LABELS) THEN
93 1         BEGIN (* THEN *)
94 2             INCLUDE(OBJ,LABELS);
95 2             WRITELN;
96 2             WRITELN(" ":10,"SEMANTIC ERROR:");
97 2             WRITE(" ":20,"LABEL ");
98 2             WRITESYM(OUTPUT,OBJ);
99 2             WRITELN(" AT TOKEN ",TOKEN:1," NOT DEFINED.")
100 2         END (* THEN *)
101 1     END (* CHECKLABEL *);
102 0
103 0 FUNCTION MAKEMAIN:SYMBOL;
104 0 (*
105 0     CREATE THE SYMBOL "TURINGOOL" AND RETURN ITS DESCRIPTOR.
106 0 *)
107 0 VAR
108 0     WASTHERE : BOOLEAN;
109 0     REP      : SYMREP;

```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11 07.32.

PAGE 4

```
110 0          STR      : UNPSTR;  
111 0          I        : INTEGER;  
112 0      BEGIN (* MAKEMAIN *);  
113 1          STR[1]:="T";  
114 1          STR[2]:="U";  
115 1          STR[3]:="R";  
116 1          STR[4]:="I";  
117 1          STR[5]:="N";  
118 1          STR[6]:="G";  
119 1          STR[7]:="O";  
120 1          STR[8]:="L";  
121 1          FOR I:=9 TO MAXCHAR DO  
122 1              STR[I]:=" ";  
123 1          PACK(STR,1,REP);  
124 1          MAKEMAIN:=HASH(REP,8,WASTHERE)  
125 1      END (* MAKEMAIN *);  
126 0  
127 0      END DECLARATIONS
```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.32.

PAGE 5

```
128 0
129 0    LANGUAGE SPECIFICATIONS
130 0
131 0    GRAMMAR ATTRIBUTES
132 0
133 0    <PROGRAM> :
134 0        ENTRY, EXIT : FGNODE;
135 0        CALLNODE : CALLPTR;
136 0
137 0    <STMT LIST> :
138 0        START : FGNODE;
139 0        FINISH : SET OF FGNODE;
140 0        LABELREF, LABELDEF : INTEGER;
141 0
142 0    <STATEMENT> :
143 0        START : FGNODE;
144 0        FINISH : SET OF FGNODE;
145 0        LABELREF, LABELDEF : INTEGER;
146 0
147 0    <DIRECTION> : ;
148 0
149 0    <IF PART> : ;
150 0
151 0    <DECLARATION> : ;
152 0
153 0    <IDENTIFIER> :
154 0        VALUE : SYMBOL;
155 0        TOKEN : INTEGER;
156 0
157 0    <STRING> :
158 0        VALUE : SYMBOL;
159 0        TOKEN : INTEGER;
160 0
161 0    <EMPTY> : ;
162 0
163 0    END GRAMMAR ATTRIBUTES
```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.32.

PAGE 6

```

164 0
165 0      RULES
166 1
167 1      <PROGRAM> ::= <DECLARATION> ";" <STMT LIST> "."
168 1      SEMANTICS
169 1
170 1      FLOWGRAPH SPECIFICATIONS
171 1
172 1          <PROGRAM>.ENTRY := NEWFGNNODE;
173 1          <PROGRAM>.EXIT := NEWFGNNODE;
174 1          <PROGRAM>.CALLNODE := NEWCGRNODE;
175 1          <STMT LIST>.LABELREF := <STMT LIST>.LABELDEF;
176 1          SETFGNTYP(<PROGRAM>.ENTRY, ENTRYSTMT);
177 1          SETFGNTYP(<PROGRAM>.EXIT, EXITSTMT);
178 1          NNEDGE(<PROGRAM>.ENTRY, <STMT LIST>.START);
179 1          SNEDGE(<STMT LIST>.FINISH, <PROGRAM>.EXIT);
180 1          SETMAINCALL(<PROGRAM>.CALLNODE);
181 1          SETCGRNAME(<PROGRAM>.CALLNODE, MAKEMAIN);
182 1          SETCGREDGE(<PROGRAM>.CALLNODE, 0);
183 1          SETCGRNTRY(<PROGRAM>.CALLNODE, <PROGRAM>.ENTRY);
184 1          SETCGREXIT(<PROGRAM>.CALLNODE, <PROGRAM>.EXIT);
185 1          SETCGRNMFP(<PROGRAM>.CALLNODE, 0);
186 1          SETCGRFPL(<PROGRAM>.CALLNODE, 0)
187 1
188 1      ACTION SPECIFICATIONS
189 1
190 1          SETACTION(DECLARE, <PROGRAM>.ENTRY, GETOBJSET(ALPHABET))
191 1      END
192 2
193 2      <STMT LIST> ::= <STATEMENT>
194 2      SEMANTICS
195 2
196 2      FLOWGRAPH SPECIFICATIONS
197 2
198 2          <STMT LIST>.LABELDEF := <STATEMENT>.LABELDEF;
199 2          <STATEMENT>.LABELREF := <STMT LIST>.LABELREF;
200 2          <STMT LIST>.START := <STATEMENT>.START;
201 2          <STMT LIST>.FINISH := <STATEMENT>.FINISH
202 2      END
203 3
204 3      <STMT LIST(1)> ::= <STMT LIST(2)> ";" <STATEMENT>
205 3      SEMANTICS
206 3
207 3      FLOWGRAPH SPECIFICATIONS
208 3
209 3          <STMT LIST(1)>.LABELDEF := <STMT LIST(2)>.LABELDEF +
210 3              <STATEMENT>.LABELDEF;
211 3          <STMT LIST(2)>.LABELREF := <STMT LIST(1)>.LABELREF;
212 3          <STATEMENT>.LABELREF := <STMT LIST(1)>.LABELREF;
213 3          <STMT LIST(1)>.START := <STMT LIST(2)>.START;
214 3          <STMT LIST(1)>.FINISH := <STATEMENT>.FINISH;
215 3          SNEDGE(<STMT LIST(2)>.FINISH, <STATEMENT>.START)
216 3      END
217 4
218 4      <STATEMENT(1)> ::= <IDENTIFIER> ":" <STATEMENT(2)>
219 4      SEMANTICS
220 4
221 4      OBJECT SPECIFICATIONS

```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.34.

PAGE 7

```

222 4
223 4      INCLUDE(<IDENTIFIER>.VALUE, LABELS)
224 4
225 4      FLOWGRAPH SPECIFICATIONS
226 4
227 4          <STATEMENT(1)>.LABELDEF := SETLABEL(<IDENTIFIER>.VALUE,
228 4              <STATEMENT(2)>.START);
229 4          <STATEMENT(2)>.LABELREF := <STATEMENT(1)>.LABELREF;
230 4          <STATEMENT(1)>.START := <STATEMENT(2)>.START;
231 4          <STATEMENT(1)>.FINISH := <STATEMENT(2)>.FINISH
232 4
233 4      ACTION SPECIFICATIONS
234 4
235 4          SETACTION(DEF, <STATEMENT(1)>.START, [ <IDENTIFIER>.VALUE ])
236 4      END
237 5
238 5  <STATEMENT> ::= "[" <STMT LIST> "]"
239 5      SEMANTICS
240 5
241 5      FLOWGRAPH SPECIFICATIONS
242 5
243 5          <STATEMENT>.LABELDEF := <STMT LIST>.LABELDEF;
244 5          <STMT LIST>.LABELREF := <STATEMENT>.LABELREF;
245 5          <STATEMENT>.START := <STMT LIST>.START;
246 5          <STATEMENT>.FINISH := <STMT LIST>.FINISH
247 5      END
248 6
249 6  <STATEMENT(1)> ::= <IF PART> <STRING> "THEN" <STATEMENT(2)>
250 6      SEMANTICS
251 6
252 6      ATTRIBUTE SPECIFICATIONS
253 6
254 6          CHECKVAR(<STRING>.VALUE, <STRING>.TOKEN)
255 6
256 6      FLOWGRAPH SPECIFICATIONS
257 6
258 6          <STATEMENT(1)>.LABELDEF := <STATEMENT(2)>.LABELDEF;
259 6          <STATEMENT(2)>.LABELREF := <STATEMENT(1)>.LABELREF;
260 6          <STATEMENT(1)>.START := NEWFGNNODE;
261 6          <STATEMENT(1)>.FINISH := [ <STATEMENT(1)>.START ] UNION
262 6              <STATEMENT(2)>.FINISH;
263 6          SETFGNTYP(<STATEMENT(1)>.START, IFSTMT);
264 6          NNEDGE(<STATEMENT(1)>.START, <STATEMENT(2)>.START)
265 6
266 6      ACTION SPECIFICATIONS
267 6
268 6          SETACTION(REF, <STATEMENT(1)>.START, [ <STRING>.VALUE ])
269 6      END
270 7
271 7  <STATEMENT> ::= <EMPTY>
272 7      SEMANTICS
273 7
274 7      FLOWGRAPH SPECIFICATIONS
275 7
276 7          <STATEMENT>.LABELDEF := 0;
277 7          <STATEMENT>.START := NEWFGNNODE;
278 7          <STATEMENT>.FINISH := [ <STATEMENT>.START ];
279 7          SETFGNTYP(<STATEMENT>.START, EMPTYSTMT)

```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.35.

PAGE 6

```

280 7      END
281 8
282 8      <STATEMENT> ::= "GO" "TO" <IDENTIFIER>
283 8      SEMANTICS
284 8
285 8      ATTRIBUTE SPECIFICATIONS
286 8
287 8      CHECKLABEL(<IDENTIFIER>.VALUE, <IDENTIFIER>.TOKEN)
288 8
289 8      FLOWGRAPH SPECIFICATIONS
290 8
291 8      <STATEMENT>.LABELDEF := 0;
292 8      <STATEMENT>.START := NEWFGNNODE;
293 8      <STATEMENT>.FINISH := [ ];
294 8      SETFGNTYP(<STATEMENT>.START, GOTOSTMT);
295 8      NNEDGE(<STATEMENT>.START,
296 8      GETLABEL(<IDENTIFIER>.VALUE, <STATEMENT>.LABELREF)
297 8
298 8      ACTION SPECIFICATIONS
299 8
300 8      SETACTION(USE, <STATEMENT>.START, [ <IDENTIFIER>.VALUE ])
301 8      END
302 9
303 9      <STATEMENT> ::= "MOVE" <DIRECTION> "ONE" "SQUARE"
304 9      SEMANTICS
305 9
306 9      FLOWGRAPH SPECIFICATIONS
307 9
308 9      <STATEMENT>.LABELDEF := 0;
309 9      <STATEMENT>.START := NEWFGNNODE;
310 9      <STATEMENT>.FINISH := [ <STATEMENT>.START ];
311 9      SETFGNTYP(<STATEMENT>.START, MOVESTMT)
312 9      END
313 10
314 10      <STATEMENT> ::= "PRINT" <STRING>
315 10      SEMANTICS
316 10
317 10      ATTRIBUTE SPECIFICATIONS
318 10
319 10      CHECKVAR(<STRING>.VALUE, <STRING>.TOKEN)
320 10
321 10      FLOWGRAPH SPECIFICATIONS
322 10
323 10      <STATEMENT>.LABELDEF := 0;
324 10      <STATEMENT>.START := NEWFGNNODE;
325 10      <STATEMENT>.FINISH := [ <STATEMENT>.START ];
326 10      SETFGNTYP(<STATEMENT>.START, PRINTSTMT)
327 10
328 10      ACTION SPECIFICATIONS
329 10
330 10      SETACTION(REF, <STATEMENT>.START, [ <STRING>.VALUE ])
331 10      END
332 11
333 11      <DIRECTION> ::= "LEFT"
334 11      SEMANTICS
335 11      END
336 12
337 12      <DIRECTION> ::= "RIGHT"

```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.35.

PAGE 9

```
338 12          SEMANTICS
339 12          END
340 13
341 13          <IF PART> ::= "IF" "THE" "TAPE" "SYMBOL" "IS"
342 13          SEMANTICS
343 13          END
344 14
345 14          <DECLARATION> ::= "TAPE" "ALPHABET" "IS" <IDENTIFIER>
346 14          SEMANTICS
347 14
348 14          OBJECT SPECIFICATIONS
349 14
350 14          INCLUDE(<IDENTIFIER>.VALUE, ALPHABET)
351 14          END
352 15
353 15          <DECLARATION(1)> ::= <DECLARATION(2)> ", " <IDENTIFIER>
354 15          SEMANTICS
355 15
356 15          OBJECT SPECIFICATIONS
357 15
358 15          INCLUDE(<IDENTIFIER>.VALUE, ALPHABET)
359 15          END
360 16
361 16          <EMPTY> ::= "NULL"
362 16          SEMANTICS
363 16          END
364 17
365 17
366 17          END RULES
367 0          END LANGUAGE SPECIFICATIONS
```

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.37.

PAGE 10

368 0
369 0 PROCEDURE SPECIFICATIONS
370 0
371 0 BEGIN
372 1 END
373 0
374 0 END PROCEDURE SPECIFICATIONS.

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.37.

PAGE 11

***** CROSS REFERENCE MAP *****

GRAMMAR SYMBOLS	RHS	LINES
<PROGRAM>	4	133 167
<STMT LIST>	3	137 167 193 204 204 238
<STATEMENT>	4	142 193 204 218 218 238 249 249 271 271
		303 314
<DIRECTION>	1	147 303 333 337
<IF PART>	5	149 249 341
<DECLARATION>	4	151 167 345 353 353
<IDENTIFIER>	0	153 218 282 345 353
<STRING>	0	157 249 314
<EMPTY>	1	161 271 361

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11 07.37.

PAGE 12

GRAMMAR ATTRIBUTES

LINES

<PROGRAM>.ENTRY	SYNTHESIZED	134	172	176	178	183	190				
<PROGRAM>.EXIT	SYNTHESIZED	134	173	177	179	184					
<PROGRAM>.CALLNODE	SYNTHESIZED	135	174	180	181	182	183	184	185	186	
<STMT LIST>.START	SYNTHESIZED	138	178	200	213	213	245				
<STMT LIST>.FINISH	SYNTHESIZED	139	179	201	214	215	246				
<STMT LIST>.LABELREF	INHERITED	140	175	199	211	211	212	244			
<STMT LIST>.LABELDEF	SYNTHESIZED	140	175	198	209	209	243				
<STATEMENT>.START	SYNTHESIZED	143	200	215	228	230	230	235	245	260	26
		263	264	264	268	277	278	279	292	294	29
		300	309	310	311	324	325	326	330		
<STATEMENT>.FINISH	SYNTHESIZED	144	201	214	231	231	246	261	262	278	29
		310	325								
<STATEMENT>.LABELREF	INHERITED	145	199	212	229	229	244	259	259	296	
<STATEMENT>.LABELDEF	SYNTHESIZED	145	198	210	227	243	258	258	276	291	30
		323									
<IDENTIFIER>.VALUE	SYNTHESIZED	154	223	227	235	287	296	300	350	358	
<IDENTIFIER>.TOKEN	SYNTHESIZED	155	287								
<STRING>.VALUE	SYNTHESIZED	158	254	268	319	330					
<STRING>.TOKEN	SYNTHESIZED	159	254	319							

RESERVED TOKENS

GO	TO	MOVE	[	]	THEN
LEFT	RIGHT	IF	ONE	SQUARE	PRINT
IS	ALPHABET	,	THE	TAPE	SYMBOL
			NULL		

SALTRAN (VERSION: 02/17/81) DATE: 81/03/05. TIME: 11.07.37.

PAGE 1

***** PROGRAM STATISTICS *****

SYMBOLS	109 (TOTAL SYMBOLS)
SPECIAL SALTRAN SYMBOLS	63
GRAMMAR/ATTRIBUTE SYMBOLS	24
OTHER SYMBOLS	22
 HASHING	
NUMBER OF CALLS	603
NUMBER OF PROBES	736
MAXIMUM PROBE	4
AVERAGE PROBE	1.22
 PRODUCTION-TABLE / PRODUCTION-LIST	50 (TOTAL ENTRIES)
PRODUCTION-TABLE	16
PRODUCTION-LIST	34
 SYNTAX RULES	16 (TOTAL)
 SEMANTIC RULES	67 (TOTAL)
OBJECT-CLASS RULES	3
ATTRIBUTE RULES	3
FLOWGRAPH RULES	56
ACTION RULES	5
OTHER RULES	0
 TABLES (PERCENT FULL)	
SYMBOL TABLE	13.6
CROSS-REFERENCE TABLE	3.3
 TOTAL PRODUCTION SYMBOLS =	9
TOTAL GRAMMAR ATTRIBUTES =	15
TOTAL RESERVED TOKENS =	22
TOTAL PROGRAM LINES =	374

MAXIMUM RIGHT-HAND-SIDE HAS 5 TERMINALS AND NONTERMINALS

TRANSLATION TIME = 4.06 SECONDS => 92.07 LINES/SECOND.

D.2 Sample TURINGOL Program

```
TREE BUILDING ANALYZER VERSION=05/13/80
TIME= 11.28.04. DATE= 81/03/05.
1  TAPE ALPHABET IS BLANK, EIN, ZERO, POINT;
12 PRINT "POINT";
15 GO TO CARRY;
19 TEST: IF THE TAPE SYMBOL IS "EIN" THEN
28     [PRINT "ZERO"; CARRY: MOVE LEFT ONE SQUARE; GO TO TEST];
44 PRINT "EIN";
47 REALIGN: MOVE RIGHT ONE SQUARE;
54 IF THE TAPE SYMBOL IS "ZERO" THEN GO TO REALIGN.
END OF ANALYSIS  COMPILE TIME= .7 SECONDS
```

D.3 Output Report for Sample Program

----- STATISTICS -----

ALLOCATED MEMORY:

	BASIC UNIT	NO. UNITS	WORDS/UNIT	TOTAL WORDS
SYMBOL TABLE	SYMBOL	32	2 (1)	64
ATTRIBUTE TABLE	ENTRY	3	1 (2)	3
PARSE TREE	NODE	47	2	94
DEPENDENCY GRAPH	NODE	155	2	310
DEPENDENCY GRAPH	EDGE	80	1	80
FLOW GRAPH	NODE	12	4 (3)	48
EXPRESSION TREE	NODE	0	4 (3)	0
CALL GRAPH	NODE	1	2	2
PRODUCTION TABLE	NUMBER	20	1	20
SET POOL	SET	47	10	470
TOTAL WORDS =				1091

- (1) INCLUDES 1 WORD(S)/STRING
 (2) INCLUDES 1 ATTRIBUTE(S)/ENTRY
 (3) INCLUDES 4 ACTION(S)/NODE

PROGRAM SIZE:

NUMBER OF PARSE TREE NODES	47		
NUMBER OF PROGRAM LINES	9	=>	5.22 NODES/LINE
NUMBER OF PROGRAM TOKENS	65	=>	0.72 NODES/TOKEN

TIMING (SECONDS):

PARSING PHASE	0.64	=>	13.95 LINES/SEC
ATTRIBUTE ANALYSIS PHASE	1.12	=>	8.04 LINES/SEC
(0.18 FOR READING TABLES AND INITIALIZING MODULES)			)
(0.09 FOR DEP. GRAPH BUILDING			)
(0.31 FOR DEP. GRAPH EVALUATION			)
(0.54 FOR DUMPING TABLES AND STATS			)
TOTAL SAM ANALYSIS TIME	1.76	=>	5.10 LINES/SEC
		=>	36.85 TOKENS/SEC

D.4 Tables File for Sample Program

```

    4  WRITE OUT ACTION NAMES
DECLARE
REF
USE
DEF
    10 WRITE OUT FLOWGRAPH NODE-TYPES
BASE
CONCAT
STRUCT
ENTRYSTMT
EXITSTMT
IFSTMT
EMPTYSTMT
GOTOCTMT
MOVESTMT
PRINTSTMT
    32 BEGINNING OF SYMBOL TABLE
    4  0 .EOF
    1  0 ;
    1  0 .
    1  0 :
    1  0 [
    1  0 ]
    4  0 THEN
    2  0 GO
    2  0 TO
    4  0 MOVE
    3  0 ONE
    6  0 SQUARE
    5  0 PRINT
    4  0 LEFT
    5  0 RIGHT
    2  0 IF
    3  0 THE
    4  0 TAPE
    6  0 SYMBOL
    2  0 IS
    8  0 ALPHABET
    1  0 ,
    4  0 NULL
    5  0 BLANK
    3  0 EIN
    4  0 ZERO
    5  0 POINT
    5  3 CARRY
    4  2 TEST
    7  1 REALIGN
    0  0
    8  0 TURINGOL
    1  1
    32  0      2      1      CALLGRAPH OVERHEAD FOR CALL-NODE 1
    0
    0
    2      4      0      END OF USE TABLE FOR CALL-NODE 1
    1      12
    0
    4      24      25      26      27
    0
    0
    0
    12      10      0
    1      11

```

1	2	
0		
1	27	
0		
0		
1	8	0
1	9	
1	12	
0		
0		
1	28	
0		
9	9	0
1	8	
2	10	11
0		
0		
1	28	
8	8	0
1	7	
1	9	
0		
0		
1	29	
0		
7	6	0
2	6	10
1	8	
0		
1	25	
0		
1	29	
10	10	0
1	9	
1	7	
0		
1	26	
0		
0		
6	10	0
1	8	
1	7	
0		
1	25	
0		
0		
5	9	0
1	3	
2	4	6
0		
0		
1	30	
3	6	0
2	1	4
1	5	
0		
1	26	
0		
0		
4	8	0

1	5	
1	3	
0		
0		
1	30	
0		
1	5	0
0		
1	3	
0		
0		
0		
J		
0		
0		
0		

END OF FLOWGRAPH FOR CALL-NODE 1
END OF EXPRESSION-TREE FOR CALL-NODE 1

D.5 User-Readable Report of Tables File

LISTING OF CALLGRAPH INFORMATION: MAINCALL= TURINGOL

NODE	EDGES
TURINGOL	:
NODE	NUM PARAM PARAM MODES
TURINGOL	: 0

DUMP OF PARTIAL FLOWGRAPH FOR SUBPROGRAM TURINGOL :

ENTRY NODE = 2, EXIT NODE = 1

----- USE-TABLE -----

----- FLOW-GRAPH -----

```

2  DESCRIPT=ENTRYSMT  EXP-TREE= 0
   SONS  = 12
   PARENTS=
     DECLARE  =                                BLANK
                                           ZERO
     REF      = <EMPTY>
     USE      = <EMPTY>
     DEF      = <EMPTY>
                                           EIN
                                           POINT
12  DESCRIPT=PRINTSMT  EXP-TREE= 0
   SONS  = 11
   PARENTS= 2
     DECLARE  = <EMPTY>
     REF      =                                POINT
     USE      = <EMPTY>
     DEF      = <EMPTY>
11  DESCRIPT=GOTOSMT  EXP-TREE= 0
   SONS  = 9
   PARENTS= 12
     DECLARE  = <EMPTY>
     REF      = <EMPTY>
     USE      =                                CARRY
     DEF      = <EMPTY>
9   DESCRIPT=MOVESMT  EXP-TREE= 0
   SONS  = 8
   PARENTS= 10 11
     DECLARE  = <EMPTY>
     REF      = <EMPTY>
     USE      = <EMPTY>
     DEF      =                                CARRY
8   DESCRIPT=GOTOSMT  EXP-TREE= 0
   SONS  = 7
   PARENTS= 9
     DECLARE  = <EMPTY>
     REF      = <EMPTY>
     USE      =                                TEST
     DEF      = <EMPTY>
7   DESCRIPT=IFSMT  EXP-TREE= 0
   SONS  = 6 10
   PARENTS= 8
     DECLARE  = <EMPTY>
     REF      =                                EIN
     USE      = <EMPTY>
     DEF      =                                TEST
10  DESCRIPT=PRINTSMT  EXP-TREE= 0
   SONS  = 9
   PARENTS= 7
     DECLARE  = <EMPTY>
     REF      =                                ZERO
     USE      = <EMPTY>
     DEF      = <EMPTY>
6   DESCRIPT=PRINTSMT  EXP-TREE= 0
   SONS  = 5
   PARENTS= 7

```

```

      DECLARE = <EMPTY>
      REF      =
      USE      = <EMPTY>
      DEF      = <EMPTY>
5  DESCRIPT=MOVESTMT  EXP-TREE= 0
   SONS  = 3
   PARENTS= 4 6
      DECLARE = <EMPTY>
      REF      = <EMPTY>
      USE      = <EMPTY>
      DEF      =
      REALIGN
3  DESCRIPT=IFSTMT  EXP-TREE= 0
   SONS  = 1 4
   PARENTS= 5
      DECLARE = <EMPTY>
      REF      =
      USE      = <EMPTY>
      DEF      = <EMPTY>
      ZERO
4  DESCRIPT=GOTOSTMT  EXP-TREE= 0
   SONS  = 5
   PARENTS= 3
      DECLARE = <EMPTY>
      REF      = <EMPTY>
      USE      =
      DEF      = <EMPTY>
      REALIGN
1  DESCRIPT=EXITSTMT  EXP-TREE= 0
   SONS  =
   PARENTS= 3
      DECLARE = <EMPTY>
      REF      = <EMPTY>
      USE      = <EMPTY>
      DEF      = <EMPTY>

```

----- EXPRESSION-TREE INFORMATION -----

SET-POOL IS 2 PERCENT FULL
0.28 SECONDS

D.6 Graphic Display of Tables File

