PB-218 875

FOLDS, A DECLARATIVE FORMAL LANGUAGE DEFINITION SYSTEM

Isu Fang

Stanford University

Prepared for:

Agency for International Development

December 1972

DISTRIBUTED BY:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

PB 218 875

FOLDS, A DECLARATIVE FORMAL LANGUAGE DEFINITION SYSTEM

BY

ISU FANG

STAN-CS-72-329
DECEMBER 1972

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

FOLDS, A DECLARATIVE FORMAL LANGUAGE DEFINITION SYSTEM

by Isu Fang

Abstract

This thesis describes FOLDS, a declarative formal language definition system. The system implements and extends Knuth's method for the specification of the semantics of context-free languages. The system provides a language (SPINDLE) and data structures to define the syntax and semantics of a language. It also includes a machine (MUTILATE) that from the definition compiles programs of the defined language. Both the consistency and the correctness of the definition can be checked in this way. The language imposes very few restrictions on definitions while preserving the declarative nature of Knuth's method; i.e , the compilation process is transparent in the definition. In addition, the system provides a means for semantically resolving syntactic ambiguities. FOLDS is intended primarily for the language designer, giving him the opportunity of realizing his definition with very little concern about implementation details. A definition of SIMULA 67 in SPINDLE and a set of SIMULA 67 programs, as compiled by the definition, are included to illustrate the capabilities of the system.

This research was supported in part by the Faculdade de Economia e Administraca da Universidade de Sao Paulo, Agency for International Development - State Department and Fundacao de Amparo a Pesquisa do Estado de Sao Paulo; by IBM Corporation; and by Xerox Corporation.

BIBLIOGRAPHIC DATA SHEET	1. Report No STAN-CS-72-329	2.	3. Recipient's Accession No.
4. Title and Subtitle Folds, A Declarative Formal Language Definition System			5. Report Date December 1972 6.
7. Author(s) Isu Fang			8. Performing Organization Rept. No. STAN-CS-72-329
9. Performing Organization Name and Address Stanford University Computer Science Department Stanford, California 94305			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address Faculdade de Economia e Administraca de Universidade de Sao Paulo; Agency for International Development-State Department and Fundacao de Amparo a Pesquisa do Estado de Sao Paulo (Brazil)			13. Type of Report & Period Covered technical 14.

15. Supplementary Notes

16. Abstracts

This thesis describes FOLDS, a declarative formal language definition system. The system implements and extends Knuth's method for the specification of the semantics of context-free languages. The system provides a language (SPINDLE) and data structures to define the syntax and semantics of a language. It also includes a machine (MUTILATE) that from the definition compiles programs of the defined language. Both the consistency and the correctness of the definition can be checked in this way. The language imposes very few restrictions on definitions while preserving the declarative nature of Knuth's method; i.e., the compilation process is transparent in the definition. In addition, the system provides a means for semantically resolving syntactic ambiguities. FOLDS is intended primarily for the language designer, giving him the opportunity of realizing his definition with very little concern about implementation details. A definition of SIMULA 67 in SPINDLE and a set of SIMULA 67 programs, as compiled by the definition, are included to illustrate the capabilities of the system.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 292
	20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70)

USCOMM-DC 40329-P71

Preface

I would like to express my deepest gratitude to my thesis advisor, Professor Donald E. Knuth, for suggesting the topic of this thesis and providing guidance and encouragement during its preparation. I also wish to thank Dr. James G. Mitchell and Professor Jerome A. Feldman for their constructive criticism, their advice and help in the preparation of this manuscript.

I also wish to thank Richard Sites, my fellow student, for many stimulating and helpful discussions and Richard E. Sweet without whose expertise and good will this thesis would never be printed.

I welcome this opportunity to thank a number of persons who through their support made this thesis possible: Professor Flavio F. Manzoli, Professor Miguel Colnago, Professor Affonso C. Pastore, Professor Sylvio Borges Reis, Vicente Paolillo and Arthur E. Angel.

This work was supported by the Faculdade de Economia e Administracao da Universidade de Sao Paulo, Agency for International Development - State Department and Fundacao de Amparo a Pesquisa do Estado de Sao Paulo. Computer time was partially provided by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183) and IBM Corporation. Preparation and publication of this paper was partially supported by IBM Corporation and by the Xerox Corporation.

I dedicate this work to my wife Sara whose love, patience, encouragement and dedication made it all possible.

TABLE OF CONTENTS

CHAPTER	SECTION		PAGE
		INTRODUCTION	1
1		REVIEW AND OVERVIEW	4
	1.1	FORMAL LANGUAGE DEFINITION METHODS	4
	1.2	IRONS' METHOD	5
	1.3	KNUTH'S METHOD	6
	1.4	A SIMULA 67 DEFINITION	13
	1.5	FOLDS	15
	1.5.1	SPINDLE - THE FOLDS LANGUAGE	18
	1.5.2	THE SPINDLE COMPILER	24
	1.5.3	MUTILATE - THE FOLDS MACHINE	25
2		SPINDLE	33
	2.1	VALUES AND CONSTANTS	36
	2.2	SYNTAX DEFINITION	37
	2.2.1	TERMINALS	37
	2.2.2	NONTERMINALS AND START SYMBOL	40
	2.2.3	SYNTACTIC PRODUCTIONS	41
	2.3	ATTRIBUTES	42
	2.3.1	ATTRIBUTE DECLARATION	46
	2.4	EXPRESSIONS	50
	2.4.1	SIMPLE EXPRESSIONS	50
	2.4.1.1	FUNCTION CALLS	52
	2.4.1.2	ATTRIBUTE DESIGNATION	53
	2.4.1.3	BLOCK EXPRESSIONS	55
	2.4.2	INTEGER EXPRESSIONS	55
	2.4.3	BOOLEAN EXPRESSIONS	57

	2.4.3.1	RELATIONS	58
	2.4.4	CONDITIONAL EXPRESSIONS	59
	2.5	STATEMENTS	60
	2.5.1	UNCONDITIONAL STATEMENTS	62
	2.5.1.1	ASSIGNMENT STATEMENTS	63
	2.5.1.1.1	OTHER RHS	68
	2.5.1.1.2	MULTIPLE ASSIGNMENTS	71
	2.5.2	CONDITIONAL STATEMENTS	73
	2.5.3	WHILE STATEMENTS	73
	2.6	OTHER EXPRESSIONS	74
	2.6.1	ASSIGNMENT EXPRESSION	74
	2.6.2	PUTIN EXPRESSIONS	75
	2.6.3	FIND EXPRESSION	77
	2.7	STANDARD FUNCTIONS	78
	2.7.1	PREDICATES	79
	2.7.2	LIST FUNCTIONS	80
	2.7.3	MISCELLANEOUS FUNCTIONS	85
	2.8	USER DEFINED FUNCTIONS AND PROCEDURES	86
	2.9	OTHER STATEMENTS	88
	2.9.1	WRITE STATEMENT	89
	2.9.1.1	FORMATED OUTPUT	90
	2.9.2	ERROR STATEMENT	93
	2.9.3	DISAMBIGUATION STATEMENT	95
	2.10	SEMANTIC RULES	97
	2.11	WRITING AND RUNNING A SPINDLE PROGRAM	99
	2.12	THE DEFINITION OF TURINGOL	100
3		MUTILATE	104
	3.1	LEXICAL ANALYZER AND PARSER	105
	3.2	INTERPRETER	107

3.3	THE INSTRUCTION SET OF MUTILATE	113
3.3.1	CONSTRUCT MANIPULATION INSTRUCTIONS	115
3.3.1.1	PLA & GET	115
3.3.1.2.	PLAN	117
3.3.1.3	GETN	117
3.3.1.4	FIND	118
3.3.1.5	FMT	118
3.3.1.6	REP	119
3.3.2	LIST MANIPULATION INSTRUCTIONS	119
3.3.2.1	CAR	120
3.3.2.2	CDR	120
3.3.2.3	CONS	121
3.3.2.4	LIST	122
3.3.2.5	APEND	122
3.3.2.6	RVRS	123
3.3.3	STACK MANIPULATION INSTRUCTIONS	123
3.3.3.1	POP	123
3.3.3.2	DBL	124
3.3.3.3	FLIP	124
3.3.4	CONTROL INSTRUCTIONS	125
3.3.4.1	JUMP	125
3.3.4.2	JUMPF & JUMPT	125
3.3.4.3	PAR & PARN	126
3.3.4.4	CALL	126
3.3.4.5	RET	127
3.3.4.6	HLT	128
3.3.4.7	ERROR	128
3.3.5	VALUE MANIPULATION INSTRUCTIONS	129
3.3.5.1	ASS	129

	3.3.5.2	TRANS	130
	3.3.5.3	VALC	130
	3.3.5.4	ASSI	131
	3.3.5.5	VAL	131
	3.3.5.6	STO	132
	3.3.5.7	LOAD	133
	3.3.5.8	AR	133
	3.3.5.9	LOG	134
	3.3.5.10	TEST	134
	3.3.5.11	COMP	135
	3.3.5.12	NAME	135
	3.3.5.13	GEN	136
	3.3.5.14	COPY	136
	3.3.6	OUTPUT INSTRUCTIONS	137
	3.3.6.1	OUT & OUTF	137
	3.3.6.2	OUTC	138
	3.3.7	THE DISAMBIGUATION INSTRUCTION - DAMB	139
	3.3.8	INDEX OF OPCODES	141
4		A DEFINITION OF SIMULA	143
	4.1	DEFINITION	145
	4.2	ANALYSIS OF THE DEFINITION	211
	4.2.1	AMBIGUITIES	212
	4.2.2	QUALTB	213
	4.2.3	VIRTUALS	213
	4.2.4	CLASS CONCATENATION	214
	4.2.5	FUNCTION INVDELTA	214
	4.2.6	CODE	215
	4.2.7	ARRAY DECLARATIONS	215
	4.2.8	LABELLED BLOCKS	216

	4.2.9	PROCEDURE AND CLASS HEADINGS	216
	4.2.10	PROCEDURE DECLARATIONS	217
	4.2.11	ST1	217
	4.2.12	OTHER MODIFICATIONS	218
5		CONCLUSION	219
		BIBLIOGRAPHY	228
		APPENDIX 1	231
		APPENDIX 2	237
		APPENDIX 3	239
		APPENDIX 4	248

LIST OF ILLUSTRATIONS

FIGURE	TITLE	PAGE
1.1	Grammar for BNI	9
1.2	Parse tree for the string 1101	9
1.3	Definition of BNI	11
1.4	Decorated parse tree for the string 1011	12
1.5	FOLDS	17
1.6	Definition of BNI in SPINDLE	23
1.7	Initial state of the machine	28
1.8	Processes to be executed	29
1.9	State 1 of the machine	30
1.10	State 2 of the machine	30
1.11	Decorated parse tree for -IO	31
2.1	Examples of elementary attributes	43
2.2	Examples of composite attributes	44
2.3	Tree representation of attributes	45
2.4	Attribute declarations	47
2.5	The attribute MATRIX	48
2.6	The attribute E(PROCDECL)	49
2.7	Declaration of attributes	66
2.8	Effect of executing assignment statements	67
2.9	Attributes after the assignments	68
2.10	Effect of the copy operator	70
2.11	Effect of CAR, CDR and CONS	81
2.12	Examples of the use of LIST functions	83
2.13	Effect of APEND	84
2.14	Effect of RVRS	84

2.15	Example of block expression	86
2.16	Examples of output statements	91
2.17	Definition using the error statement	94

INTRODUCTION

This thesis describes FOLDS, a declarative formal language definition system. The system implements and extends Knuth's method [Kn 68a] for the specification of the semantics of context-free languages: given the syntax of a language, attributes are associated with each nonterminal and the "meaning" of a string of the language is given by the values of the attributes associated with the nonterminals in the parse tree; the semantics establish, for each syntactic production, the relationships that must exist between the attributes of the nonterminals involved in the production. The system also incorporates Wilner's extensions [Wi 71] to Knuth's method.

The system provides a language (SPINDLE) and data structures to define the syntax and semantics of a language. It also includes a machine (MUTILATE) which compiles programs of the defined language using the definition. Both the consistency and the correctness of the definition can be checked in this way.

The language imposes very few restrictions on definitions while preserving the declarative nature of Knuth's method; i.e., the compilation process is transparent in the definition. In addition the system provides a means for semantically resolving syntactic ambiguities. The syntax is specified by means of productions and the semantics by means of an ALGOL-like language which serves both to relate the attributes of nonterminals as functions of other attributes and to describe the functions.

The data structure scheme is derived from the "objects" of the Vienna Definition Language [We 72] which allows great flexibility in the choice of data structures for the attributes.

The system is intended primarily for the language designer. It gives him the opportunity of realizing his definition with very little concern about implementation. With the use of MUTILATE, programs in the defined language can be compiled directly from the definition.

A large subset of SIMULA 67 has been defined in SPINDLE, both as a test for the system and as a demonstration of its capabilities; a series of SIMULA 67 programs have been compiled from this definition, the largest one being approximately 70 lines long and generating a parse tree with approximately 2000 nodes.

This thesis has been organized so that the reader can minimize the amount of reading necessary to achieve a certain depth of understanding about the system; each chapter may contain backward references but contains no forward references. The appendices are an integral part of the thesis and are used to illustrate the text.

Chapter 1 gives a general description of the system: it contains a review of formal language definition methods, with emphasis on those directly relevant to this work and an overview of FOLDS. Some simple examples illustrate the material covered. This chapter should be enough for those who only want to understand the main features and principles involved in the system.

Chapter 2 presents a description of SPINDLE, the FOLDS language. It describes the syntax and semantics of SPINDLE and gives numerous examples to illustrate its different features. A complete SPINDLE definition of a simple language is presented in Appendix 1.

This chapter should be read by those desiring a deeper understanding of the capabilities of the system and also by those who want to program in SPINDLE.

Chapter 3 describes the FOLDS machine, MUTILATE. It is essentially a terse description of the relevant aspects of the machine implementation. Appendices 2 and 3 illustrate the descriptions given in the text of the chapter. The chapter should be read only by those who want to know how some particular SPINDLE features are implemented and by those who want to implement a similar system.

Chapter 4 is a definition of a subset of SIMULA 67; it is an implementation of Wilner's definition of SIMULA 67 [Wi 71]. It illustrates both the capabilities of FOLDS and a series of SPINDLE programming techniques. Appendix 4 contains a set of SIMULA 67 programs and the target code generated for them by MUTILATE from the definition. This chapter is intended both as a demonstration to the nonbeliever of the capabilities of the system and to illustrate a series of programming techniques which may be useful for the definition of other languages. Chapter 4 presupposes an understanding of Chapter 2 but no understanding of Chapter 3.

CHAPTER 1

REVIEW AND OVERVIEW

This chapter contains a review of formal language definition methods, with emphasis on those directly relevant to this work and an overview of the FOrmal Language Definition System (FOLDS). Some simple examples will illustrate the use of the material covered.

1.1 FORMAL LANGUAGE DEFINITION METHODS

A language definition is composed of two hierarchically related sets of specifications called the syntax and semantics of the language. The syntactic component determines the set of strings that belong to the language while the semantic component attaches "meaning" to a string of the language. In particular the syntax of a programming language describes the set of valid programs and the semantics supplies the meaning of these valid programs. Much attention has been given to the problem of defining the syntax. As a result, it is well understood and has several established solutions (see for example Hopcroft & Ullman[HU 69]).

Two approaches have been used for semantic specification: interpreter-oriented and compiler-oriented. The interpreter-oriented approach defines a partial function which maps a statement and a

state vector onto a new state vector. The compiler-oriented approach, on the other hand, defines a partial function which maps a statement in the language onto a statement in another language, assumed understood.

The interpreter-oriented scheme is described by Wegner [We 72] with a detailed presentation of the Vienna Definition Language (VDL), currently the most sophisticated such method. Examples of the compiler-oriented approach appear in Irons [Ir 63], Brooker & Morris [BM 62], Wirth & Weber [WW 66], Feldman [Fe 66] and Knuth [Kn 68a].

1.2 IRONS' METHOD

Irons [Ir 63] defines the semantics of a context-free language by associating a single attribute with each non-terminal, namely its translation, and associating a semantic rule with each syntactic production. The semantic rule expresses the value of the attribute of the left hand side nonterminal (LHN) of the associated syntactic production as a function of the values of the attributes of the right hand side nonterminals (RHNs). In terms of the parse tree, a node's attribute value is determined by applying its associated semantic rule to the attribute values of its directly descendant nodes. The meaning of a string S is the attribute value attached to the root node of its parse tree , PT(S). The value of an attribute is "synthesized" from values of attributes lower in the tree. A number of compiler-compilers were based on this idea, notably McClure's [MC1 65].

1.3 KNUTH'S METHOD

Knuth [Kn 68a] extends Irons' ideas by introducing two new concepts:

(1) Multiple attributes associated with each nonterminal.

(2) Synthesized and inherited attributes.

Now the meaning of a string S is the set of values of the attributes of the root node of PT(S). The meaning of a phrase of S is the set of values of the attributes of the node from which it is derived. Synthesized attributes pass from a node to its ancestors while inherited attributes go from a node to its descendants. There are now two sets of semantic rules associated with each syntactic production. The first set establishes the values of all synthesized attributes of the LHN of the production as functions of the attributes of the RHNs together with the other attributes of the LHN. The second set establishes the values of all the inherited attributes of the RHNs of the production as a function of the attributes of the LHN and the other attributes of the RHNs. Each attribute attached to a node in the parse tree is associated with a semantic rule that establishes the attribute's value as a function of the attribute values of the surrounding nodes (ancestor, direct descendants and siblings).

The concept of multiple attributes greatly expands the meaning that can be associated with a phrase (or string). Not only the translation but any other property of a phrase (e.g. length, position on the string, etc.) can be expressed by associating attributes with the nonterminal that generates it.

Synthesized attributes are essentially like Irons' attributes. As for inherited attributes, Knuth shows that they are not essential since they can always be replaced by an equivalent set of synthesized attributes. But they greatly enhance comprehension by allowing a more natural representation, since the interplay between inherited and synthesized attributes is the way one generally thinks about such processes. Expressing language features such as labelled statements and block structure using purely synthesized attributes is complicated. Inherited attributes enable one to describe such features much more easily. In ALGOL 60, for example, the nesting depth of a block and the information about the variables which are global to it would be inherited attributes while the target code generated for the block would be a synthesized attribute. Loosely, inherited attributes represent that portion of the meaning imparted by the surrounding context of a phrase. Synthesized attributes correspond to the portion derived from the phrase itself.

Knuth introduces another concept, that of global attributes, which are attributes of the start symbol that are accessible from any production. A global attribute is equivalent to (can always be replaced by) a pair of attributes defined on all nonterminals, one synthesized and the other inherited. The synthesized attribute collects information necessary to form the value that is then propagated through the tree by the inherited attribute. This concept though not increasing the power of the method, does make the definitions written in it more concise.

One of the most important characteristics of this method is its declarative nature. The parsing method is transparent to a language definition. There is no explicit statement in a definition about the

order in which values are assigned to attributes. The semantic rules merely state how the values of the attributes of neighbouring nodes relate to each other. This contrasts with, for example, Wirth & Weber's definition of EULER which is essentially an algorithmic description.

The locality of definitions is a very important aspect of this method. The semantics of a syntactic production refer only to the values of the attributes of nonterminals involved in the production. The interdependencies between the various parts of the language are expressed only in terms of the attribute values passed between them. Besides making for more understandable and concise definitions it facilitates the addition and removal of features from the language.

As a simple example of this method we will define the binary notation for integers (BNI). The meaning of a string of 0's and 1's is its value expressed as a decimal integer. In other words we are defining the translation of binary integers to their decimal equivalents.

The grammar in figure 1.1 expresses the syntax of BNI. This grammar associates a parse tree PT(S) with any string S of BNI. The parse tree PT(1101), for the string 1101, is shown in figure 1.2.

One way of understanding binary notation is by associating values that are powers of 2 with each of the 0's and 1's. The value of the string is then the sum of the values associated with the 1's in the string. Formally:

NONTERMINALS	N, L, B, S	
TERMINALS	0 1 . -	
PRODUCTIONS	(1)	B ::= 0
	(2)	B ::= 1
	(3)	L ::= B
	(4)	L ::= L B
	(5)	N ::= S L
	(6)	S ::= .
	(7)	S ::= -
	(8)	S ::= ε
START SYMBOL	N	

COMMENTS- The nonterminals N, L, B and S stand respectively for number, list of bits, bits and sign. The symbol ε stands for the empty string and will be used throughout the report with this meaning.

Figure 1.1

Grammar for BNI

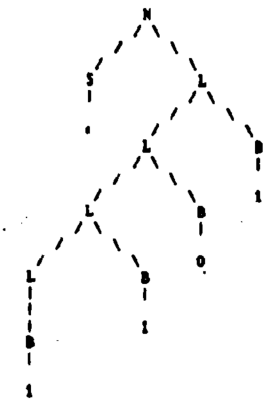


Figure 1.2

Parse tree for the string 1101

For $N = b_k b_{k-1} \ldots b_0$,

$$\text{Value}(N) = \beta_k * 2^k + \beta_{k-1} * 2^{k-1} + \ldots + \beta_0 * 2^0,$$

where β_j = if ($b_j \equiv$ "1") then 1 else 0.

In other words, the value associated with each bit in the string depends on the its location in the string. The integer attributes VALUE and SCALE associated with the nonterminal B represent respectively the value and position of a bit. These same attributes are associated with the nonterminal L: in this case VALUE stands for the sum of the values of the bits in the list of bits derived from L; SCALE for the position of the rightmost bit in the list. VALUE is also associated with N. Finally the boolean attribute NEGATIVE is associated with S, serving to convey information about the sign of the integer. VALUE and NEGATIVE are synthesized attributes, and SCALE is inherited.

With the attributes defined, semantic rules are then associated with the grammar, to express the relations between the attributes of the nonterminals of each production. This completes the definition. The rules in figure 1.3 give such a definition for BNI.

The semantic rules assume that a series of primitive notions (such as Integers, Booleans and the operations +, -, *, TRUE, FALSE and IF-THEN-ELSE) and their composition rules are well understood. In other words we are using a language which is supposedly understood to express the semantics.

The definition in figure 1.3 associates with any string S of BNI a <u>decorated</u> parse tree DPT(S) whose nodes have attributes with values assigned to them. The value of the attribute VALUE of the head

TERMINALS: 0 1 . -

ATTRIBUTES:

NAME	TYPE	KIND
VALUE	INTEGER	SYNTHESIZED
SCALE	INTEGER	INHERITED
NEGATIVE	BOOLEAN	SYNTHESIZED

NONTERMINALS:

NAME	ATTRIBUTES
N	VALUE
L	VALUE, SCALE
B	VALUE, SCALE
S	NEGATIVE

START_SYMBOL: N

PRODUCTIONS:

NUMBER	SYNTAX	SEMANTICS
(1)	B ::= 0	VALUE(B) := 0 SCALE(B) := 1
(2)	B ::= 1	VALUE(B) := 2 SCALE(B) := 1
(3)	L ::= B	VALUE(L) := VALUE(B); SCALE(B) := SCALE(L)
(4)	L ::= L B	VALUE(L) := VALUE(L*) * VALUE(B); SCALE(L*) := SCALE(L) * 1; SCALE(B) := SCALE(L)
(5)	N ::= S L	SCALE(L) := 0; VALUE(N) := IF NEGATIVE(S) THEN -VALUE(L) ELSE VALUE(L)
(6)	S ::= .	NEGATIVE(S) := FALSE
(7)	S ::= -	NEGATIVE(S) := TRUE
(8)	S ::= ε	NEGATIVE(S) := FALSE

COMMENTS- AT(NT) stands for attribute AT of nonterminal NT. An asterisk after a nonterminal identifies which occurrence of the nonterminal in the syntactic production is meant. From left to right, no asterisk corresponds to the first occurrence, one for the second, two for the third and so on.

Figure 1.3

Definition of BNI

node is the meaning of the string. An example of a decorated tree, DPT(1101), appears in figure 1.4.

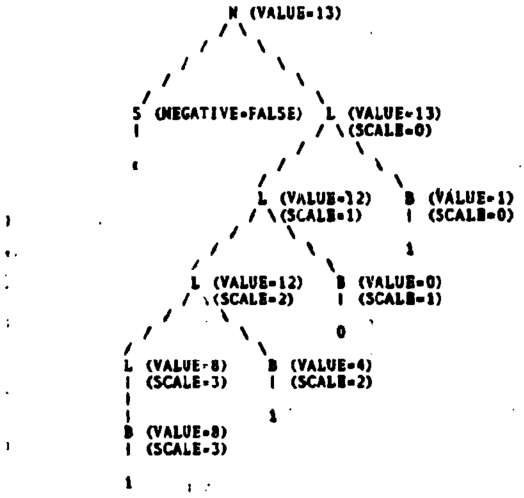


Figure 1.4

Decorated parse tree for the string 1011

The semantic rules do not define an algorithm to calculate the values of the attribute but they imply one: the attribute for the left hand side of any semantic rule can always be defined once the values that are necessary on the right hand side are all determined.

It should be noted that a string S may be syntactically correct but still have no meaning associated with it, i.e. PT(S) may exist but not DPT(S). For instance in figure 1.3 if the exponentiation function is stated to be defined only for values of the exponent that are less than 3 no meaning can be associated with strings of length greater than 3. A string S with which the definition can associate a PT(S) but not a DPT(S) is called <u>malformed</u>; if a DPT(S) can be associated it is called <u>well-formed</u>. It is the concept of well-formed

strings that allows the method to be applied to languages that are not context-free.

1.4 A SIMULA 67 DEFINITION

Using and extending Knuth's methods, Wilner [Wi 71] defines SIMULA 67. He demonstrates the method's applicability to large and complex languages by obtaining a compact and reasonably readable definition. It is only reasonably readable because the same thing happens with the SIMULA report, a reflection of the complexity of the language.

The principal extensions introduced by Wilner are called "reduction techniques". They reduce the number of semantic rules that have to be explicitly stated to define the language. The elimination of identity rules is the most important of the reduction techniques. A majority of the semantic rules are identity rules of the form $\alpha(NT_1)=\alpha(NT_2)$, where α is an attribute of both nonterminals NT_1 and NT_2 and belong to different sides of the associated syntactic production. Wilner postulates, in an informal way, that these rules do not have to be explicitly stated; they are called implicit semantic rules. The fact that α is an attribute of both NT_1 and NT_2, with no explicit semantic rule assigning a value to $\alpha(NT_1)$, implies the existence of rule $\alpha(NT_1) = \alpha(NT_2)$. Rules of this type do not

really contribute to the understanding of the semantics of a production; little is lost by not explicitly stating them, and a great conciseness of definition is gained. Wilner reports a 58% reduction in the number of rules for SIMULA 67 using this technique. Applying it, for instance, to the definition in figure 1.3 would leave production (3) with no explicit semantics and would eliminate the rule SCALE(B)=SCALE(L) from production (4).

It is interesting to observe that Wilner uses inherited and synthesized attributes but no global attributes. He argues effectively that they detract from the locality of the method and contribute very little to its conciseness, since the reduction techniques eliminate explicit rules for propagating the inherited component of the global attributes. Also the formation rules for global attributes can be very complicated, and they are easier to understand when stated step by step as synthesized attributes.

Some interesting insights into Knuth's method can be obtained from Wilner's SIMULA definition:

- Established programming language concepts such as symbol tables for block structured languages can be implemented in a very natural way; i.e., the attributes that embody these concepts and their functions reflect very closely the way one thinks about them.
- Language features which are difficult to express concisely in this method, making necessary the use of a wealth of attributes and functions for their definition (e.g., the VIRTUAL feature of SIMULA), are usually also difficult to understand and implement.
- Extensions to the language are facilitated by the

method's characteristic of locality of definition and the fact that attributes provide well defined interfaces between parts of the language (e.g. Wilner added the FOR construct to the SIMULA definition as an appendix).

The definition of SIMULA demonstrated the power of the technique but also showed that without a formal basis for the description of the semantics (i.e. a programming language) and the means to automatically check definitions it could not be considered a practical tool. The lack of a programming language to express data structures and a precise and systematic description of functions on those structures led to some ambiguous and/or incorrect definitions. (Wilner uses any convenient data structure and many of his functions are described in natural language.) Also, "hand checking" the definition proved to be an extremely painful task, due to its size and complexity. A programming language definition is an exact description of many interrelated concepts, and some mechanical checking procedure is almost mandatory because humans are notoriously bad at verifying such meticulous details.

1.5 FOLDS

The development of FOLDS makes Knuth's method a practical tool for language definition. It is a first step towards the development of compilers directly from a declarative formal definition. FOLDS provides a language (SPINDLE*) and data structures to define the

* Semantic Preparatory INput Description Language (says D. Knuth)

syntax and semantics of a language. It also provides a machine (MUTILATE*) that generates trees from this definition and fills out the associated attributes for strings of the defined language. Both the consistency and correctness of the definition can be checked in this way.

SPINDLE, the FOLDS language, imposes very few restrictions on definitions while preserving the advantages of Knuth's methods and Wilner's extensions. Both the parsing and the decoration of parse trees are completely transparent in the definition, thus preserving the declarative nature of the method.

In addition the system provides a means for semantically resolving syntactic ambiguities. It also performs syntactic checks on the definition and provides run-time error detection for easier diagnosis of definition errors.

Global attributes, as proposed by Knuth, are not provided: as noted before, Wilner does not use them because his extensions provide a viable alternative. However, the real reason for avoiding global attributes is that very few attributes are global to the whole tree in block structured languages. To be useful, the concept should be extended to resemble the global variables of ALGOL 60. An extended global would be an attribute of any nonterminal, not just the start symbol. It would be defined over any subtree derived from the nonterminal except for those subtrees where it is redefined. The inclusion of such an extended global attribute was considered, but the idea was rejected. Although more powerful than simple globals, extended globals retain some of the disadvantages which are pointed out by Wilner; furthermore the gain in conciseness could not by itself justify the significant cost of including the feature.

* Machine Underlying The Interpretive Language To be Executed (says D. Knuth)

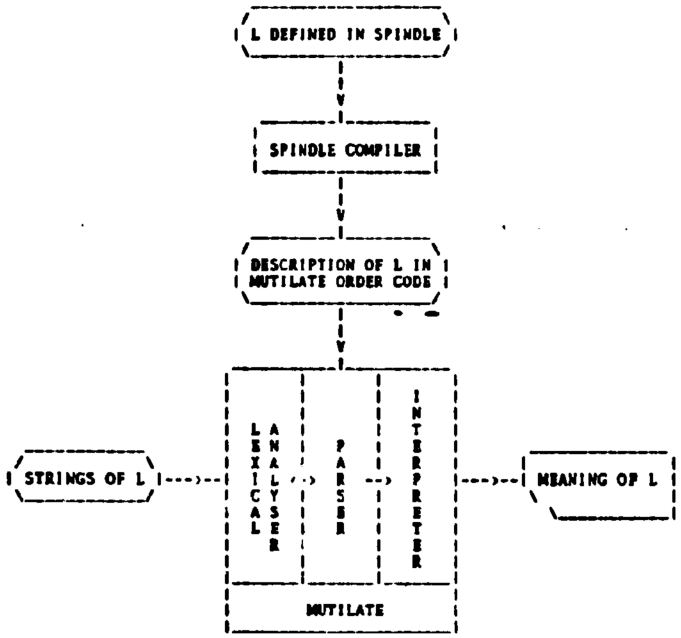


Figure 1.5

FOLDS

FOLDS is intended primarily for the language designer. It gives him the opportunity of realizing his definitions with very little concern about implementation. (While a compiler for the language is generated there need be no preocupation with efficient compilation at definition time.) It also gives him the opportunity to judge the complexity and "cost" of proposed language features.

The main benefit of the system is that the definition of a language can be stated in a well defined form. As such it can serve as a standard for the language and be understood by the users. Although not all SIMULA users will be able to understand its FOLDS

definition , those users who are capable of writing compilers will certainly be able to do so. For them it provides a precise standard against which other definitions (such as a compiler for the language) can be evaluated. Most of all, a sytem such as FOLDS imposes a discipline on the language designer that has been mostly absent in the past, making for so many unhappy language implementers.

Figure 1.5 presents a schematic view of FOLDS. The SPINDLE compiler accepts a description of a language L and compiles this description into a program in the order code of MUTILATE. This program running on MUTILATE will generate a decorated parse tree for any well-formed string of L. The following sections present brief descriptions of the components of the system.

1.5.1 SPINDLE - THE FOLDS LANGUAGE

The language is designed to give considerable flexibility to the user. It relies on a data structure representation which is derived from the objects proposed in [LLS 68], with data-types associated with them. In such an environment, composing data-types is very simple, thus facilitating the use of complex data structures.

Syntax rules are given as productions with few imposed limitations. Right and left recursion, empty strings and syntactic ambiguity are all allowed.

The syntax presupposes the existence of a lexical analyzer to handle reserved words, terminal symbols, ALGOL-like identifiers, integers, and string constants. This analyzer is a restriction on the

generality of FOLDS, but it is justified by the efficiency it brings to the system. It could of course be made more general, as in the AED system [Jo 68], with its parameters being part of the definition.

With each syntactic production is associated a number of semantic rules that manipulate the attributes of the nonterminals involved in the production. Besides the inherited and synthesized attributes, a new kind of attribute, called local attribute, is used. This attribute, whose function is to hold intermediate values, is an attribute of the head node of the corresponding production (the node of the tree associated to the LHN). It is only accessible from the semantic rules of the production. Local attributes appear both in Knuth and Wilner's work, but are used informally as an abbreviation.

Implicit semantic rules (see 1.4) do not have to be stated, being automatically generated by the system.

The language has an ALGOL flavor and incorporates features such as conditional statements and expressions, while statements, go_to statements, assignment statements, compound statements and recursive procedures.

One of the most original features of FOLDS appears in its control structure embodied in the concept of a parallel statement. A SPINDLE statement (SST) is either sequential (ST) or parallel, which is a sequence of SPINDLE statements enclosed in $/ and /$, i.e.

$$\$/\ SST_1\ ;\ SST_2\ ;\ldots;\ SST_{i-1}\ ;\ SST_i\ ;\ SST_{i+1}\ ;\ldots;\ SST_n\ /\$,\ n \geq 1.$$

SST_1 is executed after SST_{i-1} if SST_{i-1} is sequential, in parallel otherwise. For example, if we have a sequence of statements

$$\$/\ ST_1\ ;\ \$/\ ST_3\ ;\ ST_4\ /\$\ ;\ ST_2\ /\$\ ;\ ST_5$$

it will start by executing ST_1 and ST_5, complete the execution of ST_1, start ST_3 and go on immediately to execute ST_2. The execution of ST_3 (and then ST_4) goes on in parallel with the execution of ST_2; ST_5 is executed in parallel with all the others.

It should be noted that this is an unusual control structure and notation for parallelism. Usually statements are grouped to indicate that each of them is to be executed in parallel with all the others in the group; here they are grouped to indicate that they constitute an independent sequence that is to be executed in parallel with all the other statements in the program.

A process is a dynamic instance of a parallel statement. Once activated a process executes until it terminates or until it tries to access an undefined value. In the latter case the process is interrupted and passivated; it will be reactivated if and when the value is defined. All active processes run concurrently.

With each syntactic production is associated a set of parallel statements that embody the explicit semantic rules plus an implicit parallel statement to handle implicit rules (if any exist).

At run-time each node of the parse tree possesses a set of processes corresponding to the parallel statements of the production represented by the node. These processes are all activated simultaneously, possibly generating other processes. The computation ends when there are no more active processes in the system.

It should be noted that as a consequence of this structure circularities in the definition will cause the passivation of processes, that will never be reactivated since the undefined values causing the passivation depend on each other.

Another original feature of the language is the ability to resolve syntactic ambiguities by semantically "disambiguating" them. Given an ambiguous node of the tree, the proper parsing is selected by stating, in the semantic rules, the conditions which identify a particular parsing as the correct one (and all others as incorrect). This means that all possible ambiguities have to be treated by the language designer. The situation is not ideal since ambiguity is undecidable for context-free languages. On the other hand, while it is expensive, the ambiguities can be detected in practical languages. If one is present but not detected any tree which contains it will have passivated processes that will never terminate, pointing out the existence of the ambiguity. Furthermore it is not a bad idea for a language designer to be forcibly aware just how ambiguous the language being defined is and what the semantic implications of these ambiguities are. While it is widely realized that ALGOL 60 is syntactically ambiguous, the extent of this ambiguity is very often underestimated.

When the parsing tree is ambiguous the control structure operates in a slightly different fashion. A process trying to assign a value to a synthesized attribute of an ambiguous node (a node with more than one parse subtree) is passivated. If an ambiguous subtree is found to be the correct one its root node is flagged. If it is found to be incorrect it is purged; all its nodes, attributes and processes are discarded. When an ambiguous node is found to have <u>one and only one</u> correct subtree the node is disambiguated; no more processes are interrupted when trying to assign to its synthesized attributes and the ones passivated for this reason are reactivated. This control structure helps prevent the information originating from

an incorrect parsing from poisoning the rest of the parse tree, while attributes can still be synthesized and inherited in the subtrees of an ambiguous node. This is the reason why a subtree, found incorrect, can be discarded without regard to the rest of the tree.

It should be noted that some recent general purpose languages, such as NEW SAIL [Fe 72], QA4 [Di 72] and PLANNER [He 71], incorporate control structures which are somewhat similar to the ones found in SPINDLE.

A computation is well-formed if it ends with no passivated processes. Notice that a well-formed computation implies that all ambiguities have been resolved since an unresolved ambiguity would result in passivated processes. A definition is well-formed if no string will cause a computation to enter an infinite loop. Given a string S, a well-formed definition will generate a well-formed computation if S is a well-formed string of the defined language; otherwise it will generate a malformed computation. Notice that the definition may incorporate error recovery provisions. In this case a string containing errors would be a well formed string of the language whose meaning would be a set of messages indicating the errors found.

It is SPINDLE's unusual control structure that allows it to preserve the declarative nature of Knuth's method. Semantic rules state only how attributes should relate to one another without mentioning in what order values are assigned to them. They state the conditions for choosing the proper parsing without specifying the mechanism for doing it. However, SPINDLE cannot be expected to provide as primitives all the necessary functions. Auxiliary functions can be defined using the imperative elements of the

```
TERMINALS ARE  + -

RESERVED WORDS ARE 0, 1

ATTRIBUTES ARE
    VALUE = INTEGER
    SCALE = INTEGER
    COUNTER = INTEGER
    PRODUCT = INTEGER
    NEGATIVE = BOOLEAN

NONTERMINALS ARE
    N = S(VALUE)
    L = S(VALUE),I(SCALE)
    B = S(VALUE),I(SCALE)
    S = S(NEGATIVE)

  COMMENT N STANDS FOR NUMBER, L FOR LIST OF BITS, B FOR BIT AND
          S FOR SIGN;

START SYMBOL  N


$P1 B ::= 0
    $/ VALUE(B) := 0 /$

$P2 B ::= 1
    $/ COUNTER := SCALE(B);  PRODUCT := 1;
       WHILE COUNTER > 0 DO
       BEGIN
          PRODUCT := 2* PRODUCT; COUNTER := COUNTER -1
       END;
       VALUE(B) := PRODUCT /$

$P3 L ::= B
    COMMENT  NO EXPLICIT RULES;

$P4 L ::= L B
    $/ VALUE(L) := VALUE(L0) + VALUE(B) /$
    $/ SCALE(L0) := SCALE(L) + 1 /$
    COMMENT  SCALE(B) := SCALE(L) IS IMPLICIT.
             NOTICE THAT ALL 3 ASSIGNMENTS ARE EXECUTED IN PARALLEL;

$P5 N ::= S L
    $/ SCALE(L) := 0 /$
    $/ VALUE(N) := IF NEGATIVE(S) THEN -VALUE(L) ELSE VALUE(L);
       WRITE ("VALUE IS", VALUE(N)) /$
    COMMENT  NOTICE THAT IN THE SECOND PARALLEL STATEMENT THE
             ASSIGNMENT VALUE(N) := ... AND THE WRITE ARE EXECUTED
             SEQUENTIALLY;

$P6 S ::= +
    $/ NEGATIVE(S) := FALSE /$

$P7 S ::= -
    $/ NEGATIVE(S) := TRUE /$

$P8 S ::=
    $/ NEGATIVE(S) := FALSE /$
```

Figure 1.6

Definition of BNI in SPINDLE

language with local attributes performing the role of the variables of conventional languages.

A simple example of the language appears in figure 1.6. It is the definition in figure 1.3 restated in SPINDLE. The defined language uses the characters I and O (separated by blanks) instead of 1 and 0 due to the limitations of the lexical analyser. Notice that exponentiation is defined by means of a user defined function using the local attributes COUNTER and PRODUCT. To illustrate the control structure of SPINDLE, an example based on the definition in figure 1.6 is presented at the end of 1.5.3.

1.5.2 THE SPINDLE COMPILER

The compiler takes the definition of a language as input and produces a series of tables plus "object code" for the semantic rules and procedures in the order code of MUTILATE. The compiler checks the syntax, fills in implicit rules and checks for missing and illegal rules. Checks are also made to guarantee that synthesized and inherited attributes are used in the proper way and that the semantic rules of a production refer only to attributes defined for the nonterminals involved in that production.

1.5.3 MUTILATE - THE FOLDS MACHINE

When loaded with the code and tables generated by the compiler the machine reads strings of the defined language and generates the corresponding decorated parse trees (provided that the definition and strings are well-formed). It has three major parts:

- A lexical analyzer that recognizes integers, string constants (delimited by double quotes), punctuation marks, reserved words (of the defined language) and ALGOL-like identifiers. It skips over comments (which begin with the word COMMENT and end with a semicolon) and over any identifier following the reserved word END.
- A parser which interacts with the lexical analyser to build a PT(S) from an input string S. In case of ambiguity the collection of all possible PT(S)s is compactly specified.
- An interpreter which decorates PT(S) to produce DPT(S). If there is more than one PT(S) the interpreter will select the correct one using the semantic rules.

The parser is based on one presented by Fisher [Fi 70], which was itself based on Earley's [Ea 68] scheme. It has been expanded to handle strings containing empty substrings, provided that there is only a finite number of empty substrings.

This parsing scheme was chosen because it will handle any context-free language, with the exception noted above. Besides, it is efficient in the sense that, given a string of length n, in the worst case it will parse in time proportional to n^3 (ambiguous grammars),

proportional to n^2 for unambiguous grammars and proportional to n for certain classes including LR(k).

It should be noted that the constant of proportionality for this scheme is quite high and that other parsers can be more efficient. However, since their increased performance is obtained by restricting the class of grammars that they can accept they are unsuitable for FOLDS; they go against the basic philosophy of independence of definition and parsing scheme. Also, features such as syntactic ambiguity, left and right recursion, empty strings, etc., while not essential are conveniences which should be available to the user.

The interpreter mantains a multiple stack environment, one stack per process. The parallel control is implemented in a pseudo-parallel fashion with exactly one active process (called the current process) being executed at any time. A list called PROCESS (implemented as a stack) contains pointers to all other active processes. Each undefined attribute (one to whom no assignment has been made) has an associated list (implemented as a stack and called its interrupt stack), which contains pointers to those processes which have been passivated as a result of trying to access it. This list is transferred to PROCESS if and when the attribute is assigned a value. The current process may stop either because it terminated or was passivated. In the latter case, a pointer to it is placed in the interrupt stack of the attribute that caused the deactivation. The process pointed to by the top element of PROCESS is made current and the top element removed from PROCESS. When PROCESS is empty (no active processes in the system) a function DEVELOP is called and returns a node of the tree. All processes associated with this node

are then placed in PROCESS. The process pointed to by the top element is then made current and the element popped from PROCESS. On the first call DEVELOP returns the root node and in each successive call a different node, the order being a depth first traversal of the tree from left to right. When all nodes of the tree have been returned a call to DEVELOP stops the machine.

This mechanism and the control structure of SPINDLE can be illustrated by examining how the machine would handle the string - I O, given the definition in figure 1.6. The description that follows, while actually describing the mechanism, gives only the essential details and ignores allocation strategies.

Figure 1.7 indicates the state of the machine before the interpreter starts running and after the parsing of the string is completed. The tree is shown with all its attributes undefined and interrupt stacks empty. Also shown are the status of PROCESS (empty), and of LARD (LAst Returned by Develop), undefined.

In figure 1.8 each of the processes to be executed is identified, with X_{ij} standing for process j of node X_i.

The first action performed is a call to DEVELOP. A pointer to N_1 is returned, then N_{11} and N_{12} are placed in PROCESS. N_{12} is then removed from PROCESS and executed. SCALE(L_1) is assigned the value zero, its interrupt stack (empty) is placed in PROCESS (which remains unchanged) and N_{12} is terminated. Next, N_{11} is taken from PROCESS and executed. It is passivated while trying to access NEGATIVE(S_1), which is undefined; so it is placed in the NEGATIVE(S_1)

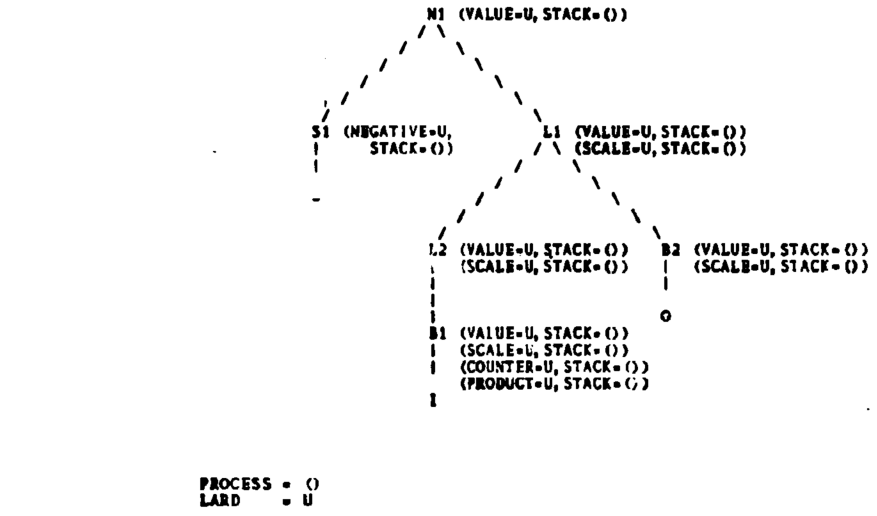


Figure 1.7

Initial state of the machine

interrupt stack. PROCESS is empty so DEVELOP is called, S_1 is returned, and S_{11} is placed in PROCESS, taken out, and executed. NEGATIVE(S_1) is assigned the value TRUE, its stack (containing N_{11}) is placed in PROCESS (which was empty) and S_{11} is terminated. N_{11} is taken out of PROCESS, executed, again passivated (this time trying to access VALUE(L_1)) and placed in VALUE(L_1)'s stack. Since PROCESS is empty, DEVELOP is called and L_{11}, L_{12} and L_{13} are placed in PROCESS. Figure 1.9shows the state of the machine at this point. L_{13} and L_{12} are then executed and terminated. L_{11} is executed, passivated (trying

```
PROCESS   DESCRIPTION

  N11     VALUE(N1) := IF NEGATIVE(S1) THEN -VALUE(L1)
                       ELSE VALUE(L1); WRITE ("VALUE IS", VALUE(N1))

  N12     SCALE(L1) := 0

  S11     NEGATIVE(S1) := TRUE

  L11     VALUE(L1) := VALUE(L2) + VALUE(B2)

  L12     SCALE(L2) := SCALE(L1) +1

  L13     SCALE(B2) := SCALE(L1)

  L21     VALUE(L2) := VALUE(B1)

  L22     SCALE(B1) := SCALE(L2)

  B11     COUNTER := SCALE(B1); PRODUCT := 1;
          WHILE COUNTER > 0 DO
          BEGIN
             PRODUCT := 2* PRODUCT;   COUNTER := COUNTER -1
          END;
          VALUE(B1) := PRODUCT

  B21     VALUE(B2) := 0
```

Figure 1.8

Processes to be executed

to access VALUE(L_2)) and placed in the interrupt stack. Next DEVELOP is called, L_2 is returned and the execution of L_{22} (terminated) and L_{21} (passivated) takes place. B_1 is then returned and B_{11} executed. During the execution, COUNTER assumes the values 1 and 0 and PRODUCT the values 1 and 2. The execution terminates after VALUE(B_1) is assigned the value 2. The state of the machine at this point is shown

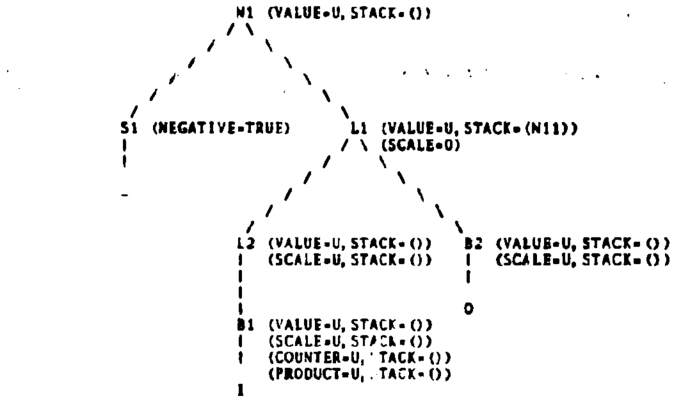


Figure 1.9

State 1 of the machine

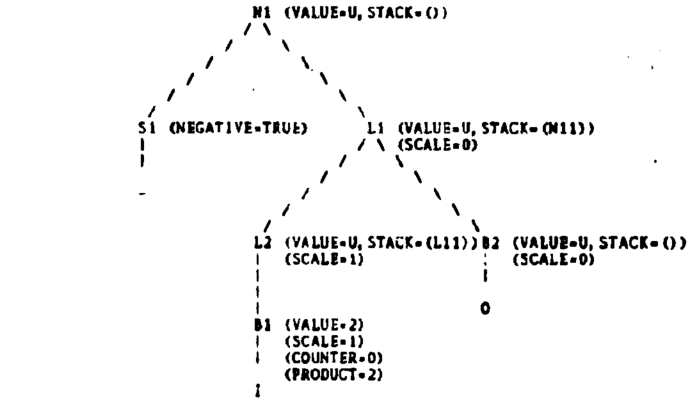
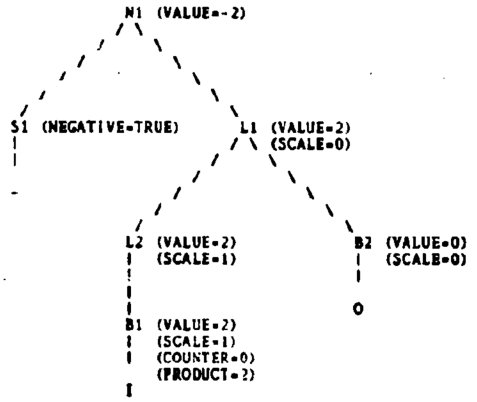


Figure 1.10

State 2 of the machine

in figure 1.10. L_{11} is reactivated and terminated. L_{21} is reactivated and passivated again, trying to access VALUE(B_2). DEVELOP is called, B_2 is returned and B_{21}, L_{21} and L_{11} are executed and terminated. Finally N_{11} is executed, VALUE(N_1) is assigned, and this is followed by the printing of the message "VALUE IS - 2" and the process is terminated. DEVELOP is called and the machine halts. Since no passivated processes remain the computation is well-formed. Figure 1.11 shows the decorated parse tree.




```
PROCESS = ()
LARD    = B2
```

Figure 1.11

Decorated parse tree for -IO

It is very important to notice that the order in which active processes are executed is entirely arbitrary. Any order can be chosen (e.g. L_{11}, S_{11}, B_{21}, L_{21}, B_{11}, L_{13}, L_{12}, N_{12}, N_{11}) and the same basic

mechanism will work successfully. The DEVELOP procedure is used only to keep the stacks from being large initially since most definitions have a left to right bias.

All that was said above is still true for ambiguous parsings; however, for implementation reasons, the order in which DEVELOP returns the nodes of the tree is not exactly the same. (For more details about the implementation of DEVELOP, see Chapter 3.)

CHAPTER 2

SPINDLE

This chapter presents a description of SPINDLE, the FOLDS language. It describes the syntax and semantics of SPINDLE and gives numerous examples to illustrate its different features. It also shows how definitions are written in SPINDLE, using TURINGOL [Kn 68a] in Appendix 1 as an example. The syntax is described using standard BNF with ϵ standing for the empty string.

SPINDLE is a metalanguage used to define languages according to Knuth's method of semantic definition. A SPINDLE program is a definition of a language according to Knuth's method; it defines the valid strings of the language and the meanings associated with them. A program when run, will recognize the well-formed strings of the defined language and associate meaning with them.

As explained in Chapter 1, the definition associates with each well-formed string S of the language a decorated parse tree DPT(S). The meaning of the string is embodied in the attributes of DPT(S)'s root node. The definition consists of a grammar plus a set of semantic rules. The grammar associates with a string S of the language a nonempty set of PT(S)s. The set is represented as a single tree with ambiguous nodes, i.e. nodes from which more than one subtree is derived. The semantic rules choose one of the PT(S)s and decorate it if S is semantically correct. In other words a string S can be syntactically correct and not be semantically correct; if

this is the case S is not a well-formed string of the defined language. This means that the method can define more than context-free languages. As shown by Floyd [Fl 62], ALGOL-60 is not a context-free language and neither is SIMULA, which is defined in Chapter 4.

The definition associates with each nonterminal a set of inherited and synthesized attributes. A node, which is a dynamic instance of a nonterminal, will then be decorated by the attributes associated with the nonterminal.

With each production of the syntax is associated a set of semantic rules that operate on the attributes of the nonterminals involved in the production. These rules serve four distinct purposes:

(1) To establish the relationship that must exist between all the inherited and synthesized attributes of the nonterminals involved in the associated production.
(2) To establish the conditions for the string to be semantically correct.
(3) To choose the right PT(S) among the set generated by the grammar.
(4) To output the values of the attributes.

The first purpose is accomplished by defining attributes as functions of other attributes; the second and third by defining predicates on the attributes; the fourth by the use of the WRITE statement. These functions and predicates are described using SPINDLE's expressions and statements, and local attributes to hold temporary values.

The scope of a local attribute consists of the semantic rules

associated with a production. Dynamically a local attribute is an attribute of the node associated with the LHN of the production. It can be manipulated only by the semantic rules associated with the node. A local attribute is attached to a node by being referenced in a semantic rule associated with the node. For example in figure 1.6 the attributes COUNTER and PRODUCT are associated with the nonterminal B of production P2 but not with the nonterminal B of production P1. This can be verified by looking at the attributes that decorate nodes B1 and B2 in figure 1.10.

The scope of the inherited and synthesized attributes associated with a node consists of the semantic rules associated with the node plus the semantic rules associated with the ancestor node. The node's semantic rules assign values to its synthesized attributes while the ancestor's rules assign values to the node's inherited attributes.

The inherited, synthesized and local attributes associated with a node are said to belong to the node.

Comments are allowed anywhere in a SPINDLE program. They begin with the reserved word COMMENT and end with a semicolon. After the reserved word END a comment may appear without the word COMMENT but may not include reserved words END, DO, or ELSE or the sequence of special characters /$.

2.1 VALUES AND CONSTANTS

The following are the primitive values of SPINDLE:

INTEGERS

STRINGS- a string of characters, enclosed in double quotes.

IDENTIFIERS- a string of letters and digits where the first character is a letter (the ALGOL identifier).

S-IDENTIFIERS- the same as IDENTIFIER but with a different internal representation.

BOOLEANS- TRUE or FALSE.

POINTERS- which are references to attributes.

COMPOSITE ATTRIBUTE VALUES- which are sets of attributes and are described in section 2.3.

TITLE- the union of STRINGS, IDENTIFIERS and S-IDENTIFIERS.

Certain of these values can be expressed by constants. The value of a constant is determined by its denotation. The syntax for constant is:

```
<CONSTANT> ::= <INTEGER> | <TITLE CONSTANT> | <BOOLEAN> |
               <POINTER CONSTANT> | <COMPOSITE ATTRIBUTE CONSTANT>

<INTEGER> ::= <DIGIT> | <INTEGER> <DIGIT>

<DIGIT> ::= 0 | 1 | 2 | .... | 8 | 9

<STRING> ::= " <... sequence of characters where a double quote is
                                denoted by a pair of double quotes...> "

<IDENTIFIER CONSTANT> ::= | <IDENTIFIER>

<IDENTIFIER> ::= <LETTER> | <IDENTIFIER> <LETTER> |
                 <IDENTIFIER> <DIGIT>
```

<LETTER> ::= A | B | C | ... | X | Y | Z

<S-IDENTIFIER> ::= & <IDENTIFIER>

<TITLE CONSTANT> ::= <S-IDENTIFIER> | <STRING> |
<IDENTIFIER CONSTANT>

<BOOLEAN> ::= TRUE | FALSE

<POINTER CONSTANT> ::= NIL

<COMPOSITE ATTRIBUTE CONSTANT> ::= NULL

NULL denotes an empty composite attribute value. NIL denotes a reference to a composite attribute whose value is NULL, whose selector is undefined and is called the null attribute.

2.2 SYNTAX DEFINITION

The syntax of the defined language is specified by defining the terminals, the nonterminals, the start symbol and the set of syntactic productions.

2.2.1 TERMINALS

The syntax pressuposes a lexical analyzer that recognises the following types of terminals: special characters, reserved words, ALGOL-like identifiers, integers and strings of characters delimited by double quotes; blanks are used as delimiters. The lexical analyzer

will skip over strings beginning with the word COMMENT and ending with a semicolon. It will ignore an identifier which follows the reserved word END. The word COMMENT may not be used either as a reserved word or as an identifier. In the defined language identifiers cannot have the same spelling as reserved words.

The following syntax is used to declare special characters and reserved words:

```
<SPECIAL CHARACTER DECLARATION> ::= ε | TERMINALS ARE
                                        <SPECIAL CHARACTER LIST>

<SPECIAL CHARACTER LIST> ::= <SPECIAL CHARACTER> |
                             <SPECIAL CHARACTER> <SPECIAL CHARACTER LIST>

<SPECIAL CHARACTER> ::= <...any special character with the
                                 exception of double quotes...>

<RESERVED WORD DECLARATION> ::= ε | RESERVED WORDS ARE
                                        <RESERVED WORD LIST>

<RESERVED WORD LIST> ::= <RESERVED WORD> |
                         <RESERVED WORD> , <RESERVED WORD LIST>

<RESERVED WORD> ::= <IDENTIFIER>
```

Terminals, other then special characters and reserved words, are handled by a SPINDLE entity called a structured terminal (S-terminal). An S-terminal is a terminal with an associated attribute; this attribute decorates all terminal nodes that are instances of the S-terminal. Identifiers, integers and strings are recognized by different S-terminals. The syntax for declaring S-terminals is:

```
<S-TERMINALS> ::= <IDENTIFIER DECLARATION> | <INTEGER DECLARATION> |
                  <STRING DECLARATION>

<IDENTIFIER DECLARATION> ::= ε | IDENTIFIERS ARE <NAME AND ATTRIBUTE>

<INTEGER DECLARATION> ::= ε | INTEGERS ARE <NAME AND ATTRIBUTE>

<STRING DECLARATION> ::= ε | STRINGS ARE <NAME AND ATTRIBUTE>
```

<NAME AND ATTRIBUTE> ::= <S-TERMINAL IDENTIFIER> WITH ATTRIBUTE
<ATTRIBUTE IDENTIFIER>

<S-TERMINAL IDENTIFIER> ::= <IDENTIFIER>

<ATTRIBUTE IDENTIFIER> ::= <IDENTIFIER>

An example of an S-terminal declaration is:

```
IDENTIFIERS ARE SIGMA WITH ATTRIBUTE SP
INTEGERS ARE NU WITH ATTRIBUTE VALUE
STRINGS ARE LAMBDA WITH ATTRIBUTE STRINGK
```

In this case an identifier, in the input string, corresponds, in the parse tree, to a node labelled SIGMA, decorated by the attribute SP whose value, in this case, is the spelling of the identifier (represented as an S-identifier value); an integer corresponds to a node NU, decorated by the attribute VALUE whose value, in this case, is the value denoted by the integer; a string corresponds to a node LAMBDA with attribute STRINGK with the string as its value.

Attribute identifiers associated with S-terminals are implicitly declared to be of kind synthesized. Attribute identifiers must be of type TITLE for identifiers and strings, and INTEGER for integers. Section 2.3 shows how to declare the attribute identifiers which will be associated with nonterminals and how to associate types with them.

2.2.2 NONTERMINALS AND START SYMBOL

The declaration of a nonterminal serves three purposes: to identify the nonterminal; to associate with it a set of inherited and a set of synthesized attribute identifiers; to associate a kind with the attribute identifier (inherited or synthesized). The syntax for nonterminal declaration is:

```
<NONTERMINAL DESCRIPTION> ::= NONTERMINALS ARE
                                  <NONTERMINAL DECLARATION LIST>

<NONTERMINAL DECLARATION LIST> ::= <NONTERMINAL DECLARATION> |
          <NONTERMINAL DECLARATION> <NONTERMINAL DECLARATION LIST>

<NONTERMINAL DECLARATION> ::= <NONTERMINAL IDENTIFIER> =
                                  <ASSOCIATED ATTRIBUTES>

<NONTERMINAL IDENTIFIER> ::= <IDENTIFIER>

<ASSOCIATED ATTRIBUTES> ::= <S-LIST> , <I-LIST> | <S-LIST> |
                            <I-LIST> , <S-LIST> | <I-LIST>

<S-LIST> =:: S ( <ATTRIBUTE LIST> )

<I-LIST> =:: I ( <ATTRIBUTE LIST> )

<ATTRIBUTE LIST> ::= <ATTRIBUTE IDENTIFIER> |
                     <ATTRIBUTE IDENTIFIER> , <ATTRIBUTE LIST>
```

An attribute identifier is declared to be of kind inherited or synthesized by appearing in an attribute list headed by an I or an S respectively.

The syntax for declaring the start symbol is:

```
<START SYMBOL DECLARATION> ::= START SYMBOL
                                  <NONTERMINAL IDENTIFIER>
```

2.2.3 SYNTACTIC PRODUCTIONS

The syntax for syntax is:

<SYNTACTIC PRODUCTION> ::= <NONTERMINAL IDENTIFIER> ::= <RIGHT HAND SIDE>

<RIGHT HAND SIDE> ::= ε | <RHS LIST>

<RHS LIST> ::= <RHS ELEMENT> | <RHS ELEMENT> <RHS LIST>

<RHS ELEMENT> ::= <SPECIAL CHARACTER> | <RESERVED WORD> | <S-TERMINAL IDENTIFIER> | <NONTERMINAL IDENTIFIER>

All special characters and identifiers appearing in a syntactic production must have been declared as such. A restriction of SPINDLE is that a right hand side of the form β$/β/$β, where β is a possibly empty sequence of RHS elements, is not allowed. An example of a syntactic production is

PROCHEAD::= IDTYPE PROCEDURE SIGMA

where given the following declarations

```
RESERVED WORDS ARE PROCEDURE
IDENTIFIERS ARE SIGMA WITH ATTRIBUTE SP
NONTERMINALS ARE
PROCHEAD = S(E)
IDTYPE = S(GENUS)
```

the production states that the string parsed from PROCHEAD is the concatenation of the string parsed from IDTYPE, followed by the reserved word PROCEDURE, and an identifier.

2.3 ATTRIBUTES

The attribute is the basic concept of SPINDLE's data structure. It is patterned after the VDL [We 72] "object" and Fisher's [Fi 70] "construct".

An attribute has a selector and a value. An attribute can then be characterized by a pair <S:V> where S is the selector and V the value. The selector names the attribute and can be either a title or an integer. If the selector is an identifier it must have been declared as an attribute identifier. For example in figure 1.11 the attributes of node L1 are <VALUE:2> and <SCALE:0>. Up to this point all attributes presented belonged to a node. But an attribute may belong to another attribute called its ancestor; i.e an attribute may have other attributes as its value. An attribute that belongs to a node is called a node attribute; if it belongs to another attribute it is called a component attribute or component for short.

Attributes may be composite or elementary. Composite attributes are those whose values are sets of attributes. Elementary attributes are those whose values are not attributes. An attribute has a type associated with it that defines its range of values. Elementary attributes can be of type INTEGER, BOOLEAN, TITLE and POINTER. In figure 2.1 are some examples of elementary attributes. ♠

Composite attributes have sets of attributes as values. Each attribute in the set is a component that belongs to the ancestor attribute. An attribute S with components $\langle S_1 : V_1 \rangle$, $\langle S_2 : V_2 \rangle$, ..., $\langle S_N : V_N \rangle$, $N \geq 0$ is represented by:

ATTRIBUTE	COMMENTS
‹SCALE: 2›	type integer; selector is an identifier.
‹"POLITICIAN": FALSE›	type boolean; selector is a string.
‹&X: DAVID›	type title; selector is an S-identifier; value is an identifier.
‹S: "SOLOMON"›	type title; selector is integer; value is a string.
‹P: @SCALE›	type pointer; selector is an integer; value is a reference to the attribute whose selector is the identifier SCALE.

Figure 2.1

Examples of elementary attributes

$$\langle S: (\langle S_1 : V_1 \rangle; \langle S_2 : V_2 \rangle; \ldots \langle S_N : V_N \rangle)\rangle$$

Composite attributes can be either of type LIST or type CONSTRUCT. The value of a construct attribute is a set of attributes with a different selector for each component. In a construct, components are referred to by their selectors. The value of a list attribute is an ordered sequence of attributes where the components have undefined selectors. In a list, components are referred to by their position in the sequence. List attributes behave exactly like their LISP [MCa 65] counterparts and are manipulated by a similar set of functions (CAR, CDR, CONS, etc.). When describing the value of a list, the components have for a selector the ordinal (parenthesized) that represent their position in the list. For example a list L with N components is described by

$$\langle L: (\langle (1); V_1 \rangle; \langle (2): V_2 \rangle; \ldots; \langle (N): V_N \rangle)\rangle$$

An empty attribute is a composite attribute whose value is the empty set; it is represented by <C: ()>, where C is any selector. An undefined attribute is an attribute whose value is undefined; it is represented here by <A: u> where A is any selector.

As an example of a composite attribute we may identify a /360 ASSEMBLER RX instruction with the construct INSTRUCTION with components OPCODE (title), R1 (integer) and OPERAND (construct). OPERAND has components D2 (title), X2 (integer) and B2 (integer). Figure 2.2a represents the instruction "A 1, LOC(2, 14)". Figure 2.2b shows the same instruction, but now associated with a list (LINSTRUCTION) instead of a construct.

```
<INSTRUCTION : (<OPCODE: A>; <R1: 1>; <OPERANDS:
                                    (<D2: LOC>; <X2: 2>; <B2: 14>)>)>
```

(a)

```
<LINSTRUCTION : (<(1): A>; <(2): 1>; <(3): (<D2: LOC>; <X2: 2>; <B2: 14>)>)>
```

(b)

Figure 2.2

Examples of composite attributes

Attributes can be conveniently represented as binary trees, the nodes representing the attributes and the edges their composition. Figure 2.3 shows the attributes defined in figures 2.1 and 2.2 in binary tree representation. An attribute is represented by a rectangle containing its value and labeled by its selector. A vertical edge connects a nonempty attribute to one of its components, called FIRST. Other components of the same attribute appear to the right of FIRST, connected by horizontal edges; the rightmost one in

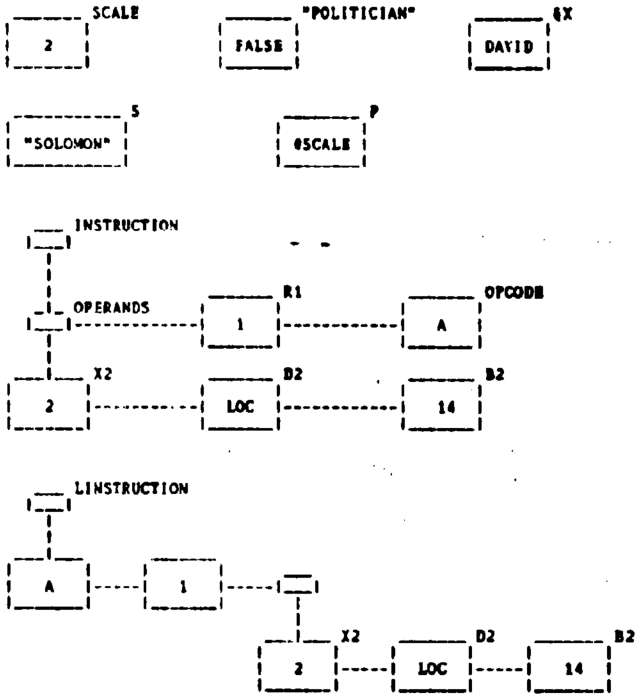


Figure 2.3

Tree representation of attributes

the sequence is called LAST. If the composite attribute is a list the order of the components from left to right reflects their position in the list; if it is a construct the order is immaterial.

Node attributes are referred to by their selectors. If the attribute is inherited or synthesized the nonterminal identifier labeling the node, parenthesized, follows the selector. For instance, INSTRUCTION refers to a local attribute with selector INSTRUCTION while SCALE(B) refers to an attribute with selector SCALE that belongs to the node B. The components of a composite attribute are

referenced through their ancestors. If the ancestor is a construct a component is referenced by prefixing its selector with a reference to the ancestor, followed by a "." . For instance, A(B).C.D refers to the component D of the component C of the attribute A which belongs to the node B. In figure 2.2a INSTRUCTION.OPERANDS.X2 is a reference to the attribute <X2:2>. If the ancestor is a list, a component is referenced by applying a composition of CAR's and CDR's to a reference to the ancestor attribute. In figure 2.2b, CAR(LINSTRUCTION) refers to the attribute <(1):A> and CAR(CDR(CDR(LINSTRUCTION))).X2 refers to <X2:2>.

2.3.1 ATTRIBUTE DECLARATION

Every attribute identifier has a type. The type of an attribute whose selector is an attribute identifier is the attribute identifier's type. An attribute identifier whose type is a construct may have an undertype. The type of a component whose selector is not an attribute identifier is its ancestor's undertype. A construct with no undertype may only have components whose selectors are attribute identifiers.

Attribute identifiers' declarations associate a type and undertype with them. Their syntax is:

```
<ATTRIBUTE DESCRIPTION> ::= ATTRIBUTES ARE
                                <ATTRIBUTE DECLARATION LIST>

<ATTRIBUTE DECLARATION LIST> ::= <ATTRIBUTE DECLARATION> |
                                <ATTRIBUTE DECLARATION>
                                <ATTRIBUTE DECLARATION LIST>
```

<ATTRIBUTE DECLARATION> ::= <ATTRIBUTE IDENTIFIER> = <ATTRIBUTE TYPE>

<ATTRIBUTE IDENTIFIER> ::= <IDENTIFIER>

<ATTRIBUTE TYPE> ::= <TYPE> | CONSTRUCT , <UNDERTYPE>

<UNDERTYPE> ::= <TYPE>

<TYPE> ::= INTEGER | BOOLEAN | TITLE | POINTER | LIST | CONSTRUCT |
<ATTRIBUTE IDENTIFIER>

When <TYPE> is an attribute identifier the type (and undertype) referred to is the type (and undertype) of the attribute identifier.

```
ATTRIBUTES ARE

    ENV = CONSTRUCT, CONSTRUCT
    E = ENV
    KIND = TITLE
    TYPE = TITLE
    CALL = TITLE
    NFORMALS = INTEGER
    CODE = POINTER
    PARAMETER = LIST
    RULE = LIST
    INSTRUCTION = CONSTRUCT
    MATRIX= CONSTRUCT, B
    B = CONSTRUCT,C
    C = CONSTRUCT, INTEGER
    P = POINTER
```

Figure 2.4

Attribute declarations

Figure 2.4 exemplifies attribute declarations. Figures 2.5 and 2.6 show examples of attributes built according to the declarations in figure 2.4. The local attribute MATRIX in FIGURE 2.5 shows how a 3-dimensional matrix can be represented as a construct and shows how components like P can be mixed with components whose type is the

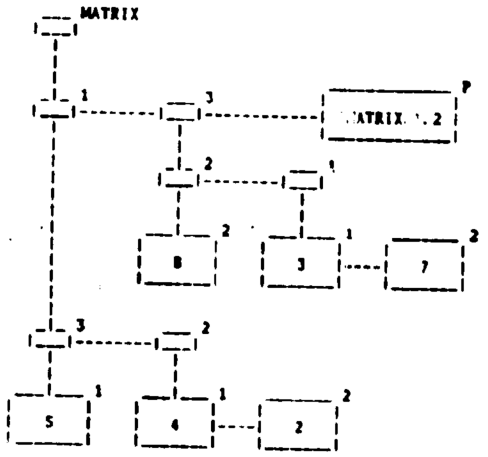


Figure 2.5

The attribute MATRIX

undertype of MATRIX. The attribute E(PROCDECL) in figure 2.6 could for example represent the symbol table built from parsing from the node PROCDECL the ALGOL procedure

```
REAL PROCEDURE MUM (X,Y); VALUE X, Y;
                          INTEGER X; REAL Y;
BEGIN
   REAL Z;
   Z: = X**Y;  X: =V+Y;
   MUM: = Z+X
END;
```

It is a consequence of this scheme for associating type with attributes that node attributes must have attribute identifiers as selectors; otherwise no type could be associated with them. For instance, in figure 2.1 only SCALE and P can be node attributes.

As seen in 2.2.2, the synthesized and inherited node attributes are defined by means of the nonterminal declarations. Attribute

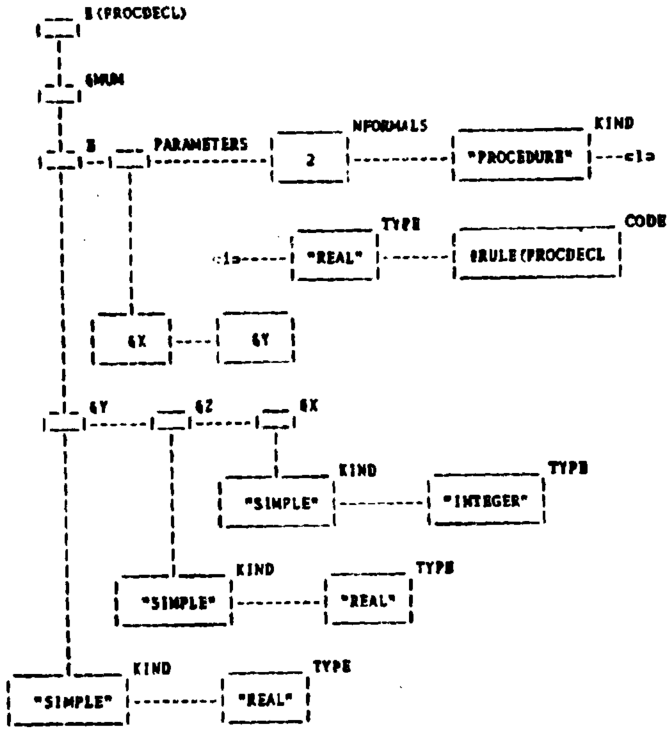


Figure 2.6

The attribute E(PROCDECL)

identifiers that do not appear in these declarations are by default of kind local.

2.4 EXPRESSIONS

SPINDLE expressions are the means for referencing attributes and manipulating their values. When evaluated, expressions return a value. The evaluation of an expression may involve the evaluation of other expressions or the execution of statements. The execution of an expression that involves an access to an undefined value will passivate the process to which the expression belongs; the process is reactivated if and when the value is defined. Their syntax is:

```
<EXPRESSION> ::= <SIMPLE EXPRESSION> | <INTEGER EXPRESSION> |
                 <BOOLEAN EXPRESSION> | <CONDITIONAL EXPRESSION>
```

2.4.1 SIMPLE EXPRESSIONS

The syntax for simple expression is:

```
<SIMPLE EXPRESSION> ::= <CONSTANT> | ( <EXPRESSION> ) |
                        [ <EXPRESSION> ] | <FUNCTION CALL> |
                        <ATTRIBUTE DESIGNATION> | <BLOCK EXPRESSION>
```

The evaluation of a constant returns the value denoted by the constant.

Parentheses enclosing an expression serve only to indicate precedence for the application of operators. The value of the parenthesized expression is the value of the expression itself.

The value resulting from the application of the bracket operator to an expression depends on the expression's value : if the expression's value is a reference to an elementary attribute, the value returned is the value of the referenced attribute; otherwise the value returned is the value of the expression (see 2.5.1.1 for further explanations). The execution of a bracketing operation will cause a passivation if the value of the operand expression is a reference to an undefined attribute.

For example, if E is an expression whose value is a reference to the attribute <SCALE:2>, the value of [E] is 2 and the value of [[E]] is also 2. If the value of E is a reference to the attribute P in figure 2.5 the value of both [E] and [[E]] is a reference to MATRIX.1.2, because this is not a reference to an elementary attribute. If the value of E is NULL the value of [E] is NULL and it it is NIL the value is NIL.

2.4.1.1 FUNCTION CALLS

A function call is composed of a function identifier and its arguments. The arguments are evaluated in sequence, from left to right; the function is then applied to the arguments and returns a value. Functions can be system defined or user defined. System defined functions are called standard functions and are described in detail in section 2.7 . The syntax for function call is:

```
<FUNCTION CALL> ::= <STANDARD FUNCTION CALL> | <USER FUNCTION CALL>
<USER FUNCTION CALL> ::= <FUNCTION IDENTIFIER>
                                        <ACTUAL PARAMETER PART>
<FUNCTION IDENTIFIER> ::= <IDENTIFIER>
<ACTUAL PARAMETER PART> ::= ε | ( <ACTUAL PARAMETER LIST> )
<ACTUAL PARAMETER LIST> ::= <ACTUAL PARAMETER> |
                            <ACTUAL PARAMETER LIST> , <ACTUAL PARAMETER>
<ACTUAL PARAMETER> ::= <EXPRESSION>
```

Section 2.8 describes the evaluation of function calls and the declaration and execution of user declared functions. CAR(LINSTRUCTION) is an example of a function call. It applies the standard function CAR to the local attribute LINSTRUCTION.

2.4.1.2 ATTRIBUTE DESIGNATION

The value of an attribute designation is a reference to an attribute. Its syntax is:

```
<ATTRIBUTE DESIGNATION> ::= <NODE ATTRIBUTE DESIGNATION> |
                            <COMPONENT DESIGNATION>

<NODE ATTRIBUTE DESIGNATION> ::= <ATTRIBUTE IDENTIFIER> |
                                 <ATTRIBUTE IDENTIFIER>
                                       ( <NONTERMINAL DESIGNATION> )

<NONTERMINAL DESIGNATION> ::= <NONTERMINAL IDENTIFIER> |
                              <NONTERMINAL DESIGNATION> *

<COMPONENT DESIGNATION> ::= <ATTRIBUTE DESIGNATION> . <COMPONENT>

<COMPONENT> ::= <ATTRIBUTE IDENTIFIER> | <TITLE CONSTANT> |
                <INTEGER> | [ <EXPRESSION> ] | <FUNCTION CALL>
```

The value of a node attribute designation is a reference to the node attribute whose selector is the attribute identifier. If the attribute identifier is followed by a parenthesized nonterminal designation, the attribute belongs to the designated node otherwise it is a local attribute. The asterisks following the nonterminal serve to distinguish between occurrences of the same nonterminal in a production. From left to right, no asterisk corresponds to the first occurrence, one for the second, two for the third and so on. If A is an attribute and NT a nonterminal, A(NT) implies that A has been declared an inherited or synthesized attribute of NT. If this is not true an error occurs. An error will also occur if NT designates a node that is not in the associated syntactic production. The attribute designation A implies that A is a local attribute, i.e, it has not been declared as either inherited or synthesized for the LHN.

The definition in figure 1.6 has examples of all the varieties of node attribute designation. The evaluation of a node attribute designation will never cause a passivation since all the attributes belonging to a node are attached to it before the processes are started. Initially all node attributes are undefined.

The value of a component designation is a reference to a component attribute whose selector is the value of <COMPONENT> and whose ancestor is the attribute referenced by the value of <ATTRIBUTE DESIGNATION>. The value of <ATTRIBUTE DESIGNATION> should be a reference to a construct (but not NIL); furthermore if the value of <COMPONENT> is not an attribute identifier the referenced construct should have an undertype. Also the value of <COMPONENT> should be either a title or an integer value. If the above conditions do not hold, an error occurs. If <COMPONENT> is an attribute identifier its value is the identifier denoted by the attribute identifier. A component designation will passivate the process associated with its execution if the ancestor does not have an attribute whose selector is the value of <COMPONENT> except when on the left hand side of an assignement (see section 2.5.1.1). The process is reactivated once the component is placed in the ancestor.

The following examples, are attribute designations in the context of the attributes represented in figures 2.5 and 2.6:

ATTRIBUTE DESIGNATION	REFERENCED ATTRIBUTE
E(PRODECL).&MUM. NFORMALS	<NFORMALS: 2>
E(PRODECL).&MUM.E. [CAR(CDR(E(PROCDECL). &MUM.PARAMETERS))].TYPE	<TYPE: "REAL">
MATRIX.1.2.1	<1: 4>
[MATRIX.P].1	<1: 4>

2.4.1.3 BLOCK EXPRESSIONS

Block expressions are patterned after the ALGOL W [Si 71] block expressions. Their syntax is:

<BLOCK EXPRESSION> ::= BEGIN <COMPOUND STATEMENT>; <EXPRESSION> END

The value of a block expression is the value of its component expression. A block expression is executed by executing first its compound statement and then evaluating its expression.

As an example of the use of block expression, in figure 1.6, the semantic rule of production P2 can be rewritten as

```
$/VALUE(B) := BEGIN
                 COUNTER := SCALE (B); PRODUCT := 1;
                 WHILE COUNTER > 0 DO
                 BEGIN
                    PRODUCT := 2* PRODUCT;
                    COUNTER := COUNTER -1
                 END;
                 PRODUCT
              END/$
```

2.4.2 INTEGER EXPRESSIONS

Integer expressions are functions from integer values to an integer value. Their syntax is:

```
<INTEGER EXPRESSION> ::= <SIMPLE INTEGER EXPRESSION>
                         <INTEGER OPERATOR> <SIMPLE EXPRESSION> |
                         - <SIMPLE EXPRESSION>

<SIMPLE INTEGER EXPRESSION> ::= <INTEGER EXPRESSION> |
```

<SIMPLE EXPRESSION>

<INTEGER OPERATOR> ::= + | - | * | / | REM

Integer expressions are evaluated from left to right; operators have no precedence over other operators, precedence is indicated by the use of parentheses. The operands of an integer operator (and of the unary -) are implicity bracketed, i.e., operands whose values are references to attributes are coerced to return the value of the attribute. Integer expressions operate on integer values ; if the coercion of an operand does not result in an integer value, an error occurs. The evaluation of an integer expression will cause a passivation if the value of an operand is a reference to an undefined attribute.

Integer operators have their usual meanings with "/" standing for integer division and REM for remainder of the integer division of the left operand by the right operand.

Examples of integer expressions can be found in figure 1.6, in productions P2 and P4. Notice that in P4, due to the implicit bracketing, the evaluation of SCALE(L) in the expression SCALE(L) +1 returns not a reference to the attribute but its value.

2.4.3 BOOLEAN EXPRESSIONS

Boolean expressions are the counterparts of integer expressions for boolean values. Their syntax is:

```
<BOOLEAN EXPRESSION> ::= <SIMPLE BOOLEAN EXPRESSION>
                         <BOOLEAN OPERATOR> <SIMPLE EXPRESSION> |
                         ¬ <SIMPLE EXPRESSION> | <RELATION>

<SIMPLE BOOLEAN EXPRESSION> ::= <BOOLEAN EXPRESSION> |
                                <SIMPLE EXPRESSION>

<BOOLEAN OPERATOR> ::= AND | OR
```

Boolean expressions are evaluated from left to right with no precedence for operators. Operands are implicitly bracketed and should have boolean values, otherwise an error occurs. If the operator is AND and the value of the left operator is FALSE the right operand is not evaluated; similarly if the operator is OR and the left operand is TRUE. A passivation occurs when the value of an operand (before the implicit brackets are applied) is a reference to an undefined boolean attribute. The operator "¬" is the negation operator.

2.4.3.1 RELATIONS

Relations are predicates that take two arguments and return a boolean value. Their syntax is:

<RELATION> ::= <SIMPLE EXPRESSION> <RELATION OPERATOR> <SIMPLE EXPRESSION>

<RELATION OPERATOR> ::= <REFERENCE RELATION OPERATOR> | <SIMPLE RELATION OPERATOR>

<REFERENCE RELATION OPERATOR> ::= == | =/=

<SIMPLE RELATION OPERATOR> ::= = | ¬= | > | ≥ | < | ≤

Relations are evaluated by evaluating first the left operand, then the right operand and then applying the operator. If the operator is a simple relational operator the operands are implicitly bracketed. Reference relation operators are used primarily to test if two references refer to the same object (==) or not (=/=). However it should noted that they can be applied to any other values since the only difference between them and relation operators is that their operators are not implicitly bracketed. Relation operators compare the values of the operands; the values should be of the same type otherwise an error ocurrs. While not an error, it is meaningless to apply the operators ≥, ≤, >, < to operands that are not integer values; the value returned, while always the same, is implementation dependent.

For the attributes represented in figures 2.5 and 2.6 we could have:

RELATION	VALUE

(E(PROCDECL).&MUM.NFORMALS + 1) > 2	TRUE
E(PROCDECL).&MUM.KIND ¬= "PROCEDURE"	FALSE
MATRIX.P == MATRIX.1.2	FALSE
[MATRIX.P] == MATRIX.1.2	TRUE
MATRIX.P = MATRIX.1.2	TRUE

2.4.4 CONDITIONAL EXPRESSIONS

Their syntax is:

<CONDITIONAL EXPRESSION> ::= <IF-CLAUSE> <EXPRESSION>
ELSE <EXPRESSION>

<IF-CLAUSE> ::= IF <EXPRESSION> THEN

The value of an if-clause is the bracketed value of its expression. This value should be boolean, otherwise an error occurs. If the expression's value is a reference to an undefined attribute the associated process is passivated. Production P5 in figure 1.6 contains an example of a conditional expression.

2.5 STATEMENTS

A statement is a unit of action. The execution of a statement is the performance of a unit of action. The execution of a statement may involve smaller units of action such as the evaluation of an expression or the execution of other statements. The syntax for statement is:

```
<PARALLEL STATEMENT> ::= $/ <SEQUENCE OF STATEMENTS> /$

<SEQUENCE OF STATEMENTS> ::= <STATEMENT> |
                             <SEQUENCE OF STATEMENTS> ; <STATEMENT>

<STATEMENT> ::= <PARALLEL STATEMENT> |
                <LABEL> : <PARALLEL STATEMENT> |
                <UNCONDITIONAL STATEMENT> | <CONDITIONAL STATEMENT> |
                <WHILE STATEMENT>

<LABEL> ::= <IDENTIFIER>
```

As explained in chapter 1, SPINDLE has parallel statements, besides the usual control structures of ALGOL-like languages. All SPINDLE statements that are not parallel statements are enclosed in a parallel statement. The execution of a parallel process involves two steps: first a process associated with it is created and activated; second the created process is executed. An active process will run until it is terminated or passivated. A process is passivated while trying to evaluate an expression involving undefined values; or while executing a function call or a procedure statement (see section 2.8); or while trying to assign a value to a synthesized attribute of an ambiguous node (see section 2.9.3). A process is reactivated if and when the value is defined; or the execution of the function or procedure is terminated; or the node is disambiguated, respectively.

If a parallel statement PST_1 contains a parallel statement PST_2 a process associated with PST_2 will be created during the execution of a process associated with PST_1. In the context of PST_1's process the execution of PST_2 is finished once the process associated with PST_2 is created and activated. The execution of PST_2 can go on without regard to the execution of PST_1's process. If PST_2 is part of a loop in PST_1, a new process is created and activated every time PST_2 is executed. The execution of a sequence of statements is then similar to the execution of a sequence of statements in ALGOL. The execution of a parallel statement in the sequence is finished once the associated process has been created and activated; the next statement in the sequence can then be executed. For example, given the sequence

$$ST_1\ ;\ \$/ST_2\ ;\ ST_3/\$;\ ST_4$$

where ST_1 and ST_4 are not go-to statements, its execution will begin with ST_1's execution followed by the creation and activation of the process associated with $\$/ST_2\ ;\ ST_3/\$$ and followed by ST_4's execution. The execution of the sequence will end once ST_4's execution is finished; the execution of the process associated with the parallel statement may or may not have terminated. For instance the process could have been passivated while executing ST_2 and this would have no

bearing in the execution of ST_4. If ST_4 were a parallel statement the sequence would be terminated once the process associated with ST_4 had been created.

Label identifiers are declared by appearing as a label of a statement. The scope of label is the smallest parallel statement, block expression, or procedure declaration that contains it.

2.5.1 UNCONDITIONAL STATEMENTS

Their syntax is:

```
<UNCONDITIONAL STATEMENT> ::= <LABEL> : <UNCONDITIONAL STATEMENT> |
                              GO TO <LABEL> | <COMPOUND STATEMENT> |
                              % <EXPRESSION> | <PROCEDURE CALL> |
                              ¢ | <ASSIGNMENT STATEMENT>

<LABEL> ::= <IDENTIFIER>
<COMPOUND STATEMENT> ::= BEGIN <SEQUENCE OF STATEMENTS> END
<PROCEDURE CALL> ::= <PROCEDURE IDENTIFIER> <ACTUAL PARAMETER PART>
```

Go-to statements change the flow of control; the statement labeled by its label is the next to be executed. The go-to statement must be in the scope of the declaration of its label or an error occurs.

The compound statement is similar to its ALGOL counterpart. Its purpose is to parenthesize a sequence of statements.

The operator "%" allows the use of an expression as a

statement. The expression is evaluated for possible side effects and its value discarded.

A procedure call is similar to a user function call with the difference that it does not return a value. Procedures are all user defined; no system defined procedures exist. Section 2.8 describes the declaration and execution of procedures and the execution of procedure calls.

2.5.1.1 ASSIGNMENT STATEMENTS

An assignment operator is applied to two operands; the L-operand (for left hand side) and the R-operand (for right hand side). The L-operand must always be a reference to an attribute, called the L-attribute; this attribute may not be the null attribute. The R-operand is either a reference to an attribute, called the R-attribute, or some other value. the assignment can take three forms depending on the type and values of the operands:

- If the R-operand is NIL or a non pointer value, it is copied into the value field of the L-attribute.
- If the R-operand is a pointer value and the L-attribute is a pointer, the R-operand is copied into the value field of the L-attribute.
- If the value of th R-operand is a non NIL pointer and the L-attribute is not a pointer, the value of the L-attribute is indirectly the value of the R-attribute which means that

the L-attribute's value is not a copy of the R-attribute's value but exactly the same value. There is no implicit copying; if desired, copying is handled explicitly (see section 2.5.1.1.1).

An attribute whose value is indirect is called an indirect attribute; otherwise it is called a direct attribute. An indirect attribute may be indirect to another indirect attribute and form a chain of indirects; at the end of an indirect chain is always a direct attribute called the final attribute. If the R-attribute is indirect the L-attribute is assigned indirectly the value of the final attribute of the R-attribute. In all cases, if the L-attribute was undefined before the statement's execution, once the assignment is complete, all processes that were passivated trying to access its value are reactivated. If the value was defined, the previous value is erased.

If the R-operand is not a pointer value, its type should be the same as the type of the L-attribute; if the L-attribute is a pointer the R-operand should be a pointer value; otherwise the type and undertype of the L-attribute and the R-attribute should be the same. If the above conditions are violated an error occurs.

The main reason for choosing this form of assignment operator is to avoid copying. Since many of the attributes used in the definition of languages are large and complex composite attributes (e.g. symbol tables) that are passed from node to node, it would not be feasible to copy the entire value of these attributes each time an assignment is made.

As a consequence of this scheme, if the value of an attribute

changes, all indirect attributes to whose indirect chains the attribute belongs, will also change. This is in a way a weakness of the SPINDLE language. Ideally the value of other inherited and synthesized attributes once assigned, should never change. This can only be accomplished by the extensive use of copying.

An indirect value is represented here by "iAD" where AD is a reference to the final attribute. For example, if the L-attribute is ‹A:u› and the R-attribute ‹B:u› the assignment will change the L-attribute to ‹A:iB›.

NOTE- Section 2.4.1. states that if an expression's value references a composite attribute the bracketing of the expression returns the same value. This is not true if the composite attribute is an indirect attribute; in this case the bracketing returns as a value a reference to the final attribute of the composite attribute.

NOTE- An attribute designation which is part of a component designation (see section 2.4.1.2) is implicitly bracketed: if the value of the ancestor attribute is indirect the component referred to is the component of its final attribute.

The syntax for assignment statements is

‹ASSIGNMENT STATEMENT› ::= ‹LHS› := ‹RHS›

‹LHS› ::= ‹ATTRIBUTE DESIGNATION›

‹RHS› ::= ‹ASSIGNMENT STATEMENT› | ‹EXPRESSION› | ‹OTHER RHS› |
‹MULTIPLE ASSIGNMENT›

An assignment statement is executed by first evaluating ‹LHS› and then ‹RHS›. The value of an assignment statement is the value of its ‹LHS›. If ‹RHS› is an assignment statement or an expression, the assignment operator is applied to the value of ‹LHS› (L-operand) and to the value of ‹RHS› (R-operand).

```
ATTRIBUTES ARE

    A = INTEGER
    A1 = A
    T = TITLE
    T1 = T
    C = CONSTRUCT, D
    C1 = C
    C2 = C1
    D = CONSTRUCT, INTEGER
    D1 = D
    P = POINTER
    P1 = P
    B = BOOLEAN
    R = CONSTRUCT, S
    S = CONSTRUCT, R
```

Figure 2.7

Declaration of attributes

The only difference between the evaluation of an attribute designation which is a ‹LHS› and one which is an expression is that the former will create components where the latter would cause a passivation. The difference occurs in a component designation where the ancestor either has an undefined value or has no component whose selector is the value of ‹COMPONENT›; if the attribute designation is an expression a passivation occurs; if it is a ‹LHS› a component is created whose selector is the value of ‹COMPONENT› and whose value is undefined. After the assignment, all processes passivated trying to

```
(a)                                    (b)

$/ A:= 2;                              <A: 2>
   T := T1 := "BRUNO";                 <T: 1T1>, <T1: "BRUNO">
   C.T1 := T;                          <C: (<T1: 1T1>)>
   C.T1 := "BOB";                      <C: (<T1: "BOB">)>
   C1 := C;                            <C1: 1C>
   D.(C.T1) := A +1;                   <D: (<"BOB": 3>)>
   C := NULL;                          <C: ()>
   C1."FRIENDS" := D;                  <C: (<"FRIENDS": 1D>)>
   C1."FRIENDS"."PAT" := 7;            <D: (<"BOB": 3>; <"PAT": 7>)>
   C2 := C1;                           <C2: 1C>
   C1 := NULL;                         <C1: ()>
   P := D."BOB";                       <P: @D."BOB">
   B := [P] == C2."FRIENDS"."BOB";     <B: TRUE>
   [P] := 4;                           <D: (<"BOB": 4>; <"PAT": 7>)>
   A := A1                             <A: 1A1>, <A1: u>
/$
```

Figure 2.8

Effect of executing assignment statements

access this component are reactivated; if the ancestor was undefined it is now defined. Due to the implicit bracketing of the attribute designation-part of a component designation, if an ancestor is an indirect attribute the new component is added to its final attribute.

The execution of the parallel statement in Figure 2.8a exemplifies the rules stated above. Figure 2.7 contains the declaration of all the attribute identifiers used in this and subsequent examples in section 2.5.1.1. Figure 2.8b shows how attributes are affected by the execution of each statement of figure 2.8a and figure 2.9 shows the status of all attributes at the end of the execution. Notice that if the last statement of figure 2.8a were A:= [A1] the process would be passivated and that instead of <A: 1A1> we would have <A: 2>.

```
<A: 1A1>
<A1: u>
<T: 1T1>
<T1: "BRUNO">
<B: TRUE>
<C: (<"FRIENDS": 1D>)>
<C1: ()>
<C2: 1C>
<D: (<"BOB": 4>; <"PAT": 7>)>
<P: @D. "BOB">
```

Figure 2.9

Attributes after the assignments

2.5.1.1.1 OTHER RHS

The syntax for assignment statements continues as follows:

```
<OTHER RHS> ::= # <EXPRESSION> | * <EXPRESSION> |
                <CONDITIONAL ASSIGNMENT>

<CONDITIONAL ASSIGNMENT> ::= <IF CLAUSE> <RHS> ELSE <RHS>
```

The "#" is the copy operator. The expression is implicitly bracketed and the value of the bracketed expression is the R-operand. If the L-attribute is not a composite attribute or the value of the R-operand is NULL the normal SPINDLE assignment takes place. Otherwise the following takes place: the value NULL is assigned to the L-attribute; then for each component of the R-attribute a

component of the same type and undertype and with the same selector and in the same order is attached to the L-attribute; then each component of the R-attribute is assigned (without copying) to the corresponding component of the L-attribute.

Notice that the expression may return a reference to the L-attribute as its value; if the L-attribute is indirect (due to the implicit bracketing of the expression) the indirectness is eliminated and the value of the final attribute copied; if the attribute is not indirect the operation has no effect on the attributes. It should also be noted that for composite attributes, while the components of the attribute are copied, if the components are themselves composite attributes, their values are not copied. It should finally be noted that for elementary attributes the bracketing of the right hand side expression has the same effect as the application of the # operator.

As an example of the copy operator the parallel statement in figure 2.10a when executed starting with the attribute in figure 2.9 will cause the changes shown in figure 2.10b.

The "*" operator creates a component of the L-attribute that is a copy of the R-attribute (same type, undertype and selector) and assigns the R-attribute to this component. For example the execution of the statement C."FRIENDS" :=* B would affect the attributes in figure 2.9 in the following way:

```
‹C: (‹"FRIENDS": iD›)›
‹D: (‹"BOB": 4›; ‹"PAT": 7›; ‹B: iB›)›
```

For an assignment involving a * operator the L-attribute should be a construct; the R-operand should be a reference to an attribute whose selector is defined; if the selector is not an attribute

```
$/  D1 :- #D;   D."ANDY" :- 9;   D1."HEATHER" :- 1;   D."BOB" :- 3;
    C2 :- #C;   C."FRIENDS" :- #C."FRIENDS";
    C."FRIENDS"."ANDY" :- 10 /$
```

(a)

```
<C: (<"FRIENDS": (<"BOB": 1D."BOB">; <"ANDY": 10>; <"PAT": 1D."PAT">)>)>
<C2: (<"FRIENDS": 1D>)>
<D: (<"BOB": 3>; <"ANDY": 9>; <"PAT": 7>)>
<D1: (<"BOB": 4>; <"PAT": 7>; <"HEATHER": 1>)>
```

(b)

Figure 2.10

Effect of the copy operator

identifier, the type of the R-attribute must be the same as the undertype of the L-attribute. If the above conditions are not satisfied an error occurs.

The conditional assignment chooses one of its <RHS> to be the <RHS> of the assignment statement. If the value of the if-clause is TRUE the leftmost <RHS> is used, otherwise the rightmost one is used.

2.5.1.1.2 MULTIPLE ASSIGNMENTS

The multiple assignment operator "$" is SPINDLE's counterpart of VDL's -operator. It allows a single statement to assign values to different components of an attribute. Its syntax is:

```
<MULTIPLE ASSIGNMENT> ::= $ ( <COMPONENT ASSIGNMENT SEQUENCE> )

<COMPONENT ASSIGNMENT SEQUENCE> ::= <COMPONENT ASSIGNMENT> |
          <COMPONENT ASSIGNMENT SEQUENCE> ; <COMPONENT ASSIGNMENT>

<COMPONENT ASSIGNMENT> ::= <COMPOUND COMPONENT> := <RHS> |
                           <PARALLEL COMPONENT ASSIGNMENT> |
                           <CONDITIONAL COMPONENT ASSIGNMENT>

<COMPOUND COMPONENT> ::= <COMPONENT> |
                         <COMPOUND COMPONENT> . <COMPONENT>

<PARALLEL COMPONENT ASSIGNMENT> ::= $/ <COMPONENT ASSIGNMENT> /$

<CONDITIONAL COMPONENT ASSIGNMENT> ::= <IF CLAUSE>
                                           <COMPONENT ASSIGNMENT>
                                       ELSE <COMPONENT ASSIGNMENT>
```

The effect of executing a component assignment

```
<COMPONENT PART> := <RHS>.
```

which is part of a multiple assignment

```
<LHS> := $(...)
```

is the same as the effect of executing the assignment statement

```
<LHS>.<COMPONENT PART> := <RHS>.
```

For example, the multiple assignment statement

```
R. "KELSON" := $("RUTH".A := 23;
                 "DORIS" := $(A := 20; T := "JOE"))
```

and the sequence of statements

```
R."KELSON"."RUTH".A := 23; R."KELSON". "DORIS". A:=20;
R. "KELSON. "DORIS".T:= "JOE"
```

when executed have exactly the same effect upon the environment. The parallel component assignment allows the execution of the component assignment as a separate process , i.e. in parallel with the rest of the multiple assignment. It is equivalent to the associated assignment statement being a parallel statement. The multiple assignment is executed from left to right in exactly the same order that the associated compound statement would be executed. For example, given the attribute <R:u>, the execution of the statement

```
$/R."KELSON" := $("RUTH".A := R."KELSON"."DORIS".A + 3;
                  "DORIS".A :=20)/$
```

would cause the associated process to passivate trying to evaluate R."KELSON"."DORIS".A and result in the attribute

<R:(<"KELSON":(<"RUTH":(<A:u>)>)>)>.

If no other parallel statement assigns a value to R."KELSON"."DORIS".A the process will never be reactivated. On the other hand, under the same circumstances, the execution of

```
$/R."KELSON" := $($/"RUTH".A := R."KELSON"."DORIS".A +3/$;
                  "DORIS".A := 20)/$
```

would generate two process that when terminated would result in the attribute

<R: (<KELSON: (<RUTH: (<A: 23>); <DORIS: (<A: 20>)>)>)>

2.5.2 CONDITIONAL STATEMENTS

<CONDITIONAL STATEMENT> ::= <LABEL> : <CONDITIONAL STATEMENT> |
<IF STATEMENT> |
<IF STATEMENT> ELSE <STATEMENT>

<IF STATEMENT> ::= <IF CLAUSE> <UNCONDITIONAL STATEMENT>

The conditional statement has exactly the same control structure as its ALGOL counterpart. As in the ALGOL conditional statement, it is possible to execute the unconditional statement without evaluating the if-clause by using the GO TO statement.

2.5.3 WHILE STATEMENTS

<WHILE STATEMENT> ::= <LABEL> : <WHILE STATEMENT> |
WHILE <EXPRESSION> DO <STATEMENT>

The control structure of the WHILE statement is similar to its ALGOL W counterpart. The expression is implicitly bracketed and returns a boolean value. Unlike ALGOL W, it is possible not to evaluate the expression the first time around by transfering directly

to the statement by means of a GO TO statement. In figure 1.6, the semantic rule of production P2 contains an example of a while statement.

2.6 OTHER EXPRESSIONS

Section 2.4 presents an incomplete syntax for SPINDLE expressions. The following are the missing forms:

‹EXPRESSION› ::= ‹ASSIGNMENT EXPRESSION›
‹SIMPLE EXPRESSION› ::= ‹PUTIN EXPRESSION› | ‹FIND EXPRESSION›

2.6.1 ASSIGNMENT EXPRESSION

The assignment expression is a form of ‹EXPRESSION› not mentioned in section 2.4. Its syntax is:

‹ASSIGNMENT EXPRESSION› ::= ‹ATTRIBUTE IDENTIFIER› ** ‹RHS›

The only difference between the execution of an assignment statement and the evaluation of an assignment expression is in the evaluation of the left hand side. In the assignment expression, the

L-attribute is a new attribute, called an <u>isolated attribute</u>, that does not belong to a node or an attribute; the attribute identifier establishes the type and selector of the isolated attribute. The expression's value is a reference to the isolated attribute. Notice that since the isolated attribute is not a node attribute or a component, the only way to refer to it is by means of the reference returned by the evaluation of the expression. For example, the execution of the parallel statement

```
$/ A := 2;  P1 := A *= 3;
      A1 := A *= A + [P1] +4;  A := A1 + [P1] /5
```

results in the local attributes ‹A:12›, ‹A1:iA$_2$› and ‹P1:@a$_1$› and in the isolated attributes ‹A$_1$:3› and ‹A$_2$:9›.

Assignment expressions are extremely useful inside iterative statements where for each iteration a new attribute has to be created. An example of this use is shown in section 2.7.2.

2.6.2 PUTIN EXPRESSIONS

The purpose of the PUTIN expression is to insert new components into a construct. Its syntax is:

```
‹PUTIN EXPRESSION› ::=  PUTIN ( ‹ATTRIBUTE DESIGNATION› :
                                ‹COMPONENT ASSIGNMENT SEQUENCE› )
```

The execution of a PUTIN expression is equivalent to the execution of the assignment statement

<ATTRIBUTE DESIGNATION> := $(<COMPONENT ASSIGNMENT SEQUENCE>)

with the following differences:

- The attribute designation is evaluated as an expression, not as a LHS.
- The value of a PUTIN expression is a reference to the attribute referred to by the simple attribute designation part of the attribute designation.

For example, given the attribute

<R: (<"KELSON" : (<"RUTH": (<A: 23>)>)>)>

the execution of

PUTIN (R."KELSON", "DORIS".A := 20 ; "BRUNO".A := 17)

would return as a value a reference to the attribute

<R: (<"KELSON": (<"RUTH": (<A: 23>)>; <"BRUNO": (<A: 17>)>;
<"DORIS": (<A: 20>)>)>)>

It should be noted that the equivalent assignment statement

R."KELSON" := $("BRUNO".A:=17 ; "DORIS". A:= 20)

would return as a value a reference to the attribute "KELSON" (if it were the RHS of another assignment). Also notice in the above example that if R were undefined, the PUTIN expression would passivate while the equivalent assignment statement would not. In other words PUTIN only adds to attributes already defined. Finally it should be noted that if the component assignment sequence has parallel parts they go on asynchronously; i.e., PUTIN may be done before they are finished. The attribute designation part of the PUTIN expression should return a reference to a construct (but not a NIL value) or an error occurs.

2.6.3 FIND EXPRESSION

A find expression is used to check the presence of a certain component in a construct. Its syntax is:

‹FIND EXPRESSION› ::= FIND (‹EXPRESSION› , ‹COMPONENT›)

The value of ‹EXPRESSION› should be a reference to a construct or NIL (which is a reference to a construct with value NULL), otherwise an error occurs. If a construct has a component whose selector is the value of ‹COMPONENT› the expression's value is a reference to the component; otherwise the value is NIL. As a consequence, if the expression's value is NIL the value of FIND is NIL. The evaluation of the FIND expression will cause a passivation if the construct is undefined. For example, given ‹R:u›, the execution of the parallel statement

```
$/    R."KELSON" := $("RUTH".A := 23;  "DORIS".A := 20);
      P := FIND(R."KELSON", "DORIS");
      P1 := FIND(R."KELSON", "BRUNO")/$
```

results in

```
<R: (<"KELSON": (<"RUTH": (<A: 23>)>; <"DORIS": (<A: 20>)>)>)>
<P: @R."KELSON"."DORIS">
<P1: NIL>
```

However, it should be noted that given the parallel statements

```
$/    R."KELSON" := $("RUTH".A := 23; "DORIS".A := 20) /$
$/    P := FIND(R."KELSON", "DORIS") /$
```

after both are executed and terminated the value of the attribute P is either @R."KELSON"."DORIS" or NIL. This can be avoided by replacing the first parallel statement by

```
$/    R := R1."KELSON" := $( ... ) /$
```

In this case R is undefined until the complete construct is assigned and P will always be assigned the value @R."KELSON"."DORIS".

2.7 STANDARD FUNCTIONS

Standard functions are system defined functions that complement the operators furnished by the language. A standard function is evaluated by first evaluating its arguments from left to right and

then applying the function to the values returned by the arguments. The value returned by a standard function varies from function to function. Their syntax is:

```
<STANDARD FUNCTION> ::= <PREDICATES> | <LIST FUNCTIONS> |
                        <MISCELANEOUS FUNCTIONS>
```

2.7.1 PREDICATES

A predicate's value is always boolean. Their syntax is:

```
<PREDICATES> ::= NULLR ( <EXPRESSION> ) |
                 NULLB ( <EXPRESSION> )
```

The value of NULLR is TRUE if the value returned by the expression is either FALSE, 0, NULL, or NIL; otherwise it is FALSE. The value of NULLB is TRUE if the expression's value is either FALSE, 0, NULL, NIL or if it is a reference to an attribute whose value is either FALSE, 0, NIL, or NULL; otherwise it is FALSE. NULLB will cause a passivation if the value of the expression is a reference to an undefined attribute. For example, given the attribute <C: ()> the value of NULLR(C) is FALSE while the value of NULLB(C) is TRUE.

2.7.2 LIST FUNCTIONS

List functions are used to manipulate lists. The value of a list function is either a reference to a list component or a special kind of list called a value-list. A value-list is a list whose selector is undefined and that does not belong either to a node or to another attribute. When a value list is the R-operand of an assignment, the value assigned to the L-attribute (which must be a list) is directly the value of the value-list. If the R-operand were a reference to a list and if the L-attribute were also a list the L-attribute's value would indirectly be the value of the R-attribute. Notice that if the L-attribute is a pointer and the R-operand a value list, an error occurs. The value list is a list and not a reference to a list. The syntax for list function is:

```
<LIST FUNCTION> ::= CAR ( <EXPRESSION> ) |
                    CDR ( <EXPRESSION> ) |
                    CONS ( <EXPRESSION> , <EXPRESSION> ) |
                    LIST ( <EXPRESSION> ) |
                    APEND ( <EXPRESSION> , <EXPRESSION> ) |
                    RVRS ( <EXPRESSION> )
```

The functions CAR, CDR, CONS, and LIST correspond exactly to their LISP counterparts and work essentially in the same way. As in LISP, the list components are not copied and the application of these functions to a list does not change its value.

CAR takes a value-list or a reference to a list as an argument and returns a reference to its first component. An error occurs if the expression's value is not a a value-list or a reference to a list or if the list or the value list is empty. For example, given the list <L; (<(1):3>;<(2):4>)>, CAR(L) returns a reference to <(1):3>.

CDR takes a value-list or a reference to a list as an argument and returns a value-list whose components are all the components of the argument list but the first. If the value of the argument list is NULL an error occurs. For example, figure 2.11 shows the list L and L1 before and after the execution of the statement L1 := CDR(L). Notice that the value of L1 is direct and that no copy was performed.

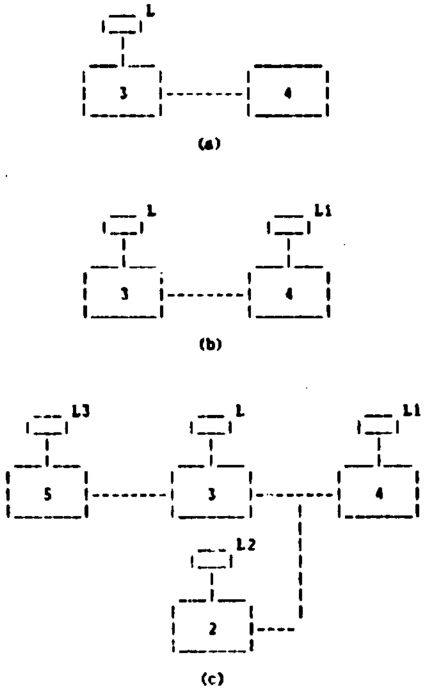


Figure 2.11

Effect of CAR, CDR and CONS

The application of the CONS function creates a new attribute whose type and value are determined by the value of the first argument : if the argument has a nonreference value or is not NULL or NIL the new attribute has the appropiate type to receive the value;

if it is a reference to an attribute, the new attribute has the same type and undertype as this attribute; if it is NIL or NULL it is a pointer with value NIL. In all cases the new attribute has an undefined selector. After the creation of the new attribute an assignment is performed with the new attribute as the L-attribute and the first argument as the R-operand. The second argument is a value list or a reference to a list. The value of CONS is a value-list whose first component is the new attribute and whose other components are those of the second argument's list. For example the execution of the sequence of statements

```
L2 := CONS(2,L1) ; L3 := CONS( CAR(L1) + CAR(L2) , L)
```

transforms the attributes in figure 2.11b into the attributes in figure 2.11c.

The execution of the function LIST(ARG) is always equivalent to the execution of CONS(ARG, L**=NULL).

Figure 2.12 is an example of the use of the list functions. The execution of the compound statement (a) transforms the attributes (b) into the attributes (c). Observe that in line 5 of the compound statement, the attribute designation COUNTER is bracketed; if not, the value of ADDRESS would be iCOUNTER in both $INSTRUCTION_1$ and $INSTRUCTION_2$.

The functions APEND and RVRS differ from the other list functions in that they change the value of the list upon which they are applied. They correspond to the LISP functions APPEND and REVERSE with the difference that the LISP functions do not change the values

```
BEGIN
   L2 := NULL;  COUNTER := 1;  L1 := L;
   WHILE ~NULLB(L1) DO
   BEGIN
      L2 := CONS(INSTRUCTION ** $(ADDRESS := CAR(L1);
                                  OP1 := (COUNTER)), L2);
      L1 := CDR(L1);  COUNTER := COUNTER +1
   END
END
```

(a)

<L: (<(1): 31>; <(2): 42>)>

(b)

<L1: ()>

<L: (<(1): 31>; <(2): 42>)>

<INSTRUCTION$_1$: (<OP1: 1CAR(L)>; <ADDRESS: 1>)>

<INSTRUCTION$_2$: (<OP1: 1CAR(CDR(L))>; <ADDRESS: 2>)>

<L2: (<(1): 1INSTRUCTION$_1$>; <(2): 1INSTRUCTION$_2$>)>

(c)

Figure 2.12

Examples of the use of LIST functions

of their argument lists. The reason for using APEND and RVRS is their greater efficiency, both timewise (no sequence of CARs and CDRs as in APPEND) and spacewise (no new attributes are created). The arguments of both APEND and RVRS should be either value-lists or references to lists (but not NIL) or an error occurs.

The value of APEND is a value-list whose components are the components of the first argument followed by the components of the second argument. The components of the second argument also follow

the components of all lists whose last component was the last component of the first argument. For example, figure 2.13 shows the result of executing the statement L5 := CONS(7, APEND(L, L4)) given the attributes in figure 2.11c and <L4: (<(1):6>)>. APEND should be used with extreme care since it can form circular lists which can then cause a process to enter an infinite loop.

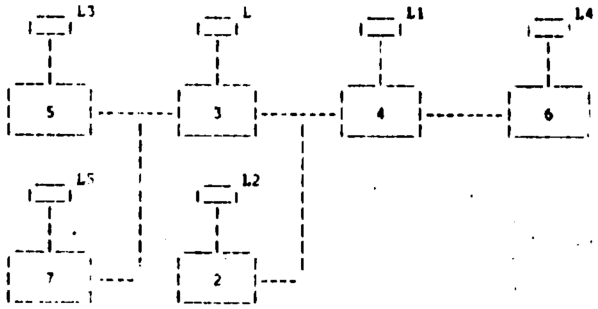


Figure 2.13

Effect of APEND

The value of RVRS is a value-list whose components are in the reverse order in which they were in the argument; the reversal affects all lists to which this components belong. Figure 2.14 shows the result of executing L3 := RVRS(L) given the attribute in figure 2.11c.

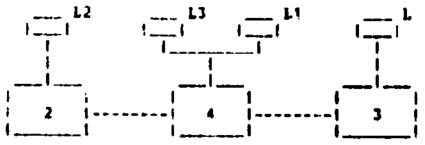


Figure 2.14

Effect of RVRS

A list function will cause a passivation if any of its arguments is a reference to an undefined list.

2.7.3 MISCELLANEOUS FUNCTIONS

```
<MISCELLANEOUS FUNCTIONS> ::= NEWINTEGER |
                              SELECTOR ( <EXPRESSION> ) |
                              FIRST ( <EXPRESSION> ) |
                              NEXT ( <EXPRESSION> )
```

The function NEWINTEGER returns a different integer value for each call on the function.

The argument of SELECTOR should be a reference to an attribute whose selector is defined, otherwise an error occurs. The value of the function is an integer if the selector is an integer, otherwise it is a title value. For example, given the attributes <A:5> and <P:@A> the value of SELECTOR(P) is P and of SELECTOR([P]) is A.

The argument of the function FIRST should be a reference to a construct, otherwise an error occurs. If the construct is undefined a passivation occurs. The value of the function is a reference to the component FIRST (see section 2.3). If the construct is empty the value of the function is NIL.

The argument of the function NEXT should be a reference to an attribute, otherwise an error occurs. If the referenced attribute is a component the value of NEXT is a reference to the component that follows the one referenced by the argument; if the referenced

component is LAST then the value of NEXT is NIL. If the referenced attribute is a node attribute, the value of NEXT depends on the implementation. (On MUTILATE, NEXT will return a reference to another attribute of the same node (or NIL)).

The block expression in figure 2.15 illustrates the use of these functions. Given C, a construct, and P, a pointer, the block expression returns the same value as FIND(C,&X).

```
BEGIN
   P := FIRST(C);
   WHILE ¬NULLB(P) DO
   IF SELECTOR((P)) = &X THEN GO TO EXIT ELSE P := NEXT((P));
EXIT: ;
   (P)
END
```

Figure 2.15

Example of block expression

2.8 USER DEFINED FUNCTIONS AND PROCEDURES

The declaration of user defined functions and procedures follows the syntax:

```
<PROCEDURE DESCRIPTION> ::= ε | <PROCEDURE DECLARATION> |
                  <PROCEDURE DECLARATION> ; <PROCEDURE DESCRIPTION>

<PROCEDURE DECLARATION> ::= FUNCTION <FUNCTION IDENTIFIER>
                             <FORMAL PARAMETER PART> ; <EXPRESSION> |
                            PROCEDURE <PROCEDURE IDENTIFIER>
                             <FORMAL PARAMETER PART> ; <STATEMENT>

<FUNCTION IDENTIFIER> ::= <IDENTIFIER>

<PROCEDURE IDENTIFIER> ::= <IDENTIFIER>
```

<FORMAL PARAMETER PART> ::= ε | (<FORMAL PARAMETERS>)

<FORMAL PARAMETERS> ::= <ATTRIBUTE IDENTIFIER> |
<FORMAL PARAMETERS> , <ATTRIBUTE IDENTIFIER>

A function or procedure call is executed as follows:

(1) The actual parameters are evaluated from left to right.
(2) A node is created and attributes whose selectors are the formal parameters are attached to it.
(3) Each actual parameter (R-operand) is assigned to the attribute whose selector is the corresponding formal parameter (L-attribute). If the number of formal and actual parameters is not the same, an error occurs.
(4) The process from wich the call was made is passivated and a process, corresponding to the body of the function or procedure, is created and activated.
(5) Once the process is terminated the calling process is reactivated and if call was a function call the value of the expression is returned.

All the node attributes used in the procedure or function body belong to the node associated with the procedure or function, thus they must all be local. The procedure body is implicitly parallel so that the scope of all the labels declared in it is the body itself.

For example in figure 1.6 the exponentiation could have been declared as:

```
FUNCTION EXP (COUNTER);
BEGIN
   PRODUCT := 1;
   WHILE COUNTER > 0 DO
   BEGIN
      PRODUCT := 2 * PRODUCT;
      COUNTER := COUNTER - 1
   END;
   PRODUCT
END;
```

The semantic rule of production P2 would then be

```
$/ VALUE(B) := EXP(SCALE(B)) /$
```

2.9 OTHER STATEMENTS

Besides the statements shown in 2.5, SPINDLE has three other types of statements:

```
<STATEMENT> ::= <WRITE STATEMENT> | <ERROR STATEMENT> |
                <DISAMBIGUATION STATEMENT>
```

2.9.1 WRITE STATEMENT

The write statement is the means for outputing values in SPINDLE. Its syntax is:

```
<WRITE STATEMENT> ::= <LABEL> : <WRITE STATEMENT> |
                      WRITE ( <OUTPUT LIST> )
<OUTPUT LIST> ::= <OUTPUT ELEMENT> | <OUTPUT LIST> , <OUTPUT ELEMENT>
<OUTPUT ELEMENT> ::= <EXPRESSION> | /
```

The statement is executed by evaluating, in sequence, from left to right, each output element. The implementation of the system guarantees that values that follow one another in the output list will follow one another in the printed output, unless the evaluation of an output element causes a passivation. No passivation occurs if the output element is an expression that references an undefined attribute. The implementation also guarantees that an attribute containing an undefined value is printed either when the value is defined or when the computation terminates (no more active processes). The implementation also guarantees that if the execution of a write statement follows the execution of another write statement (with other types of statements possibly being executed in between), the printed output of the former immediately follows the printed output of the latter. No sequencing is possible among the output lists generated by different processes. In chapter 3 it can be seen how this was implemented in MUTILATE.

Values are printed following one another in the same output line until the line is full. Once full, a line is printed and a new

one is started. The control character "/" forces the printing of the line currently being filled and starts a new one.

If the output element is an expression, it is implicitly bracketed and the value returned determines what is to be printed: if the value is a non pointer value or NIL the value is printed; otherwise the selector (if defined) and value of the referenced attribute are printed.

Integers are printed in left justified form. Strings are printed without the surrounding double quotes. If the value of a pointer attribute is not NIL the selector of the referenced attribute preceded by the character "#" is printed; otherwise NIL is printed. Composite attributes are printed by printing each of its components; the components are separated by commas and the whole list is enclosed in parenthesis. Figure 2.16 shows a series of examples of write statements and the resulting output. Notice that the components of a construct are printed in the same order they are internally stored (which depends on the implementation).

2.9.1.1 FORMATED OUTPUT

Constructs can be printed in a "nicer" way than described above, if they have a format attribute as a component. Format attributes are title attributes whose selector is FORMAT and whose value is a format identifier. The construct to which the format attribute belongs is printed according to the format associated with the format identifier. Formats are associated with format identifiers

```
STATEMENT:
        WRITE ( "VALUE IS =", CONS(6, CONS(4, CONS(2, LIST(0)))))
OUTPUT:
VALUE IS =(6, 4, 2, 0)


STATEMENT:
        $/  I := 3;  B := 6TITLE;
            C := $(A := 3;  B1 := B;  C1 := $(TYPE := "INTEGER";
                                              KIND := "ARRAY"));
            WRITE(B, C, I) /$
OUTPUT:
B= TITLE   C= (B1= TITLE, C1=(TYPE= INTEGER, KIND= ARRAY), A= 3)
I= 3
```

Figure 2.16

Examples of output statements

by means of declarations. Format attributes are attached to constructs by means of format assignments. A format assignment is a form of component assignment. Its syntax is:

<COMPONENT ASSIGNMENT> ::= <FORMAT ASSIGNMENT>
<FORMAT ASSIGNMENT> ::= FORMAT := <FORMAT IDENTIFIER>

Formats can also be attached as any other component. For example, the three following statements have exactly the same effect:

```
C: = $(FORMAT := F3)
C: = $(FORMAT := ! F3)
C. FORMAT := ! F3
```

The syntax for format declaration is:

```
<FORMAT DESCRIPTION> ::= ( | FORMATS ARE <FORMAT DESCRIPTION LIST>

<FORMAT DESCRIPTION LIST> ::= <FORMAT DECLARATION> |
                              <FORMAT DECLARATION>
                                        <FORMAT DESCRIPTION LIST>

<FORMAT DECLARATION> ::= <FORMAT IDENTIFIER> =
                                        ( <FORMAT ELEMENT LIST> )

<FORMAT IDENTIFIER> ::= <IDENTIFIER>

<FORMAT ELEMENT LIST> ::= <FORMAT ELEMENT> |
                          <FORMAT ELEMENT> , <FORMAT ELEMENT LIST>

<FORMAT ELEMENT> ::= / | <ATTRIBUTE IDENTIFIER> ! <STRING>
```

An example of a format declaration is:

```
F4= (OPER,"(",OP1,",",OP2,")",/,"GO-TO(",LABEL,")")
```

The format controls the printing by executing in succession, from left to right, each of the format elements; if the format element is a string the string is printed. If it is a "/", the line being filled is printed; if it is an attribute identifier, the value of the component whose selector is the identifier is printed; if no such component exists nothing is printed. The selector of the composite attribute to which the format attribute belongs is not printed. As an example, with F4 declared as above, the parallel statement

```
$/  C:= $ ( OPER := "ADD";  OP1 := 1;  OP2 := 5;
            TYPE := "RR";  FORMAT := F4;  LABEL := "EXIT");
    WRITE (C) /$
```

will print

```
ADD(1,5)
GO-TO(EXIT)
```

The same statement without the format assignment would print

C= (OPER = ADD, OP2 = 5, OP1 = 1, LABEL = EXIT, TYPE = RR)

2.9.2 ERROR STATEMENT

The error statement is one of the means by which malformed strings are detected in SPINDLE. Its syntax is:

<ERROR STATEMENT> ::= ERROR (<OUTPUT LIST>) |
<LABEL> : <ERROR STATEMENT>

The error statement prints the output list and then passivates all active processes, ending the computation.

The definition in figure 2.17 shows an example of the use of the error statement. Given a base, a sign and an integer number in this base (represented by a string of integers), the definition will output the decimal value of the number. Notice that if the base is greater than 9 or if the number contains an improper digit the string is malformed.

```
TERMINALS ARE  + -

ATTRIBUTES ARE
    VALUE • INTEGER
    SCALE • INTEGER
    BASE • INTEGER
    COUNTER • INTEGER
    PRODUCT • INTEGER
    NEGATIVE • BOOLEAN

INTEGERS ARE NU WITH ATTRIBUTE VALUE

NONTERMINALS ARE
    N • S(VALUE)
    L • S(VALUE), I(SCALE)
    S • S(NEGATIVE)

START SYMBOL  N


FUNCTION EXP(BASE, COUNTER, VALUE)
BEGIN
   IF VALUE ≥ BASE THEN
   ERROR (VALUE, " IS NOT VALID FOR NUMBERS BASE", BASE);
   PRODUCT :• 1;
   WHILE COUNTER › 0 DO
   BEGIN
      PRODUCT :• PRODUCT • BASE;
      COUNTER :• COUNTER -1
   END;
   PRODUCT • VALUE
END


$P1 L ::• NU
    $/VALUE(L) :• EXP(BASE(L), SCALE(L), VALUE(NU)) /$

$P2 L ::• L NU
    $/ VALUE(L) :• VALUE(L•) •
                    EXP(BASE(L), SCALE(L), VALUE(NU)) /$
    $/ SCALE(L•) :• SCALE(L) • 1 /$

$P3 N ::• NU S L
    $/ SCALE(L) :• 0 /$
    $/ BASE(L) :• VALUE(NU) /$
    $/ IF VALUE(NU) › 9 THEN
                    ERROR (VALUE(NU), "IS NOT A PROPER BASE");
       VALUE(N) :• IF NEGATIVE(S) THEN -VALUE(L) ELSE VALUE(L);
       WRITE ("VALUE IS", VALUE(N)) /$

$P4 S ::• •
    $/ NEGATIVE(S) :• FALSE /$

$P5 S ::• -
    $/ NEGATIVE(S) :• TRUE /$
```

Figure 2.17

Definition using the error statement

2.9.3 DISAMBIGUATION STATEMENT

The disambiguation statement is the means for handling ambiguities in SPINDLE. Its syntax is:

```
<DISAMBIGUATION STATEMENT> ::= <LABEL> : <DISAMBIGUATION STATEMENT> |
                               DAMB ( <EXPRESSION> , <NODE> )
<NODE> ::= <INTEGER> | <NONTERMINAL IDENTIFIER>
```

Every process is associated with a nonterminal node of the parsing tree called the process's node. For function and procedure bodies this node is the node associated with the calling process. An ambiguous node sprouts more than one parsing subtree. An ambiguous node is disambiguated if one and only one of its subtrees is correct.

The function of the disambiguation statement, as the name implies, is to check for correct parsings. The expression in the first operand is implicitly bracketed and returns a boolean value (or an error occurs). If the value is TRUE the subtree to which the current node belongs and whose root is the node designated by the second operand is the correct parsing; if it is FALSE, it is an incorrect one. If the second operand is an integer the designated node is the ith ambiguous node in the ancestor line of the process's node, starting with the process's node itself. For example, if the process's node is ambiguous, a "1" for the second operand refers to the process's node and a "2" to its first ambiguous ancestor. If the integer in the second operand designates a nonexistent ambiguous node an error occurs. If the second operand is a nonterminal identifier, the designated node is the first ambiguous node in the ancestor line,

starting with the process's node, that is labelled by the identifier; if no such node exists an error occurs.

The synthesized attributes of an ambiguous node can only be assigned values after the node is disambiguated; processes trying to assign values to the node before disambiguation are passivated. If a subtree is found incorrect it is discarded together with all its attributes and processes. If a subtree is found correct, it is kept. After all parsings of a node have been checked, if more than one correct parsing is found an error occurs; if only one is correct, the node is disambiguated and all passivated processes trying to assign to its synthesized attributes are reactivated. If no parsing is correct then:

(1) if the ambiguous node has no ambiguous ancestor an error occurs;
(2) if it has ambiguous ancestors the subtree attached to the nearest ancestor that contains this node is marked incorrect.

Notice that if an ambiguous node is not detected or if one of the possible subtrees of an ambiguous node is not recognized as such, the processes trying to assign to the synthesized attributes of the node will be passivated and will never terminate.

The use of the disambiguation statement is illustrated in section 2.12 when the definition of TURINGOL is discussed.

2.10 SEMANTIC RULES

As explained in Chapter 1, a set of semantic rules is associated with each syntactic production. The semantic rules operate on the attributes of the nodes involved in the production. Certain semantic rules are implied, i.e they do not have to be explicitly stated, being automatically generated by the system. It is a characteristic of this method of semantic definition that the semantic rules of a production can only assign to the synthesized attributes of the LHN, the inherited attributes of the RHNs and to local attributes. It is an error to assign to an inherited attribute of the LHN or a synthesized attribute of a RHN. SPINDLE introduces the restriction that no inherited or synthesized attribute of a node can appear in the left hand side of an assignment statement more than once in the semantic rules associated with a production; if this happens, an error occurs. For example, in production P5 of figure 1.6, it would be an error to write

```
IF NEGATIVE(S) THEN VALUE (N) := -VALUE (L)
               ELSE VALUE (N) :=  VALUE (L);
```

and it would also be wrong to write

```
$/ IF NEGATIVE(S) THEN VALUE (N) := VALUE (L) / $
$/ IF ¬NEGATIVE(S) THEN VALUE (N) := VALUE (L) / $
```

Implicit semantic rules are always of the form $A(NT_1) = A(NT_2)$ where A is an attribute and NT_1 and NT_2 nonterminals on opposite sides of a production. Given the production

$$L ::= R_1 R_2 \ldots R_i \ldots R_n$$

if an inherited attribute I, belonging to R_i, does not appear as a left hand side of any assignment in the associated semantic rules, and if I also belongs to L, the rule $I(R_i) := I(L)$ is automatically generated; if I is not an attribute of L an error occurs. If a synthesized attribute S of L does not appear as a left hand side of any assignment in any of the associated semantic rules, and if S is an attribute of R_i the semantic rule $S(L) := S(R_i)$ is generated; if S is an attribute of more than one RHN or of none of them, an error occurs.

Semantic rules are organized into parallel statements. Semantic rules whose values depend on one another, have to be either in different parallel statements or, in a sequence of statements, the dependent one has to come after the one it depends on. For instance, the semantic rules of production P4 in figure 1.6 could have been written as

```
$/ SCALE (L*) := SCALE (L) + 1 ;
   VALUE (L) := VALUE (L*) + VALUE (B) /$
```

However, if the order of the statements in this parallel statement were reversed, the process would never terminate. Therefore separate parallel statements should ordinarily be used for each attribute.

Productions and their associated semantic rules are described by the following syntax:

```
<PRODUCTION DESCRIPTION> ::= <PRODUCTION> |
                             <PRODUCTION> <PRODUCTION DESCRIPTION>

<PRODUCTION> ::= $ <LABEL> <SYNTACTIC PRODUCTION> <SEMANTIC RULES>

<SEMANTIC RULES> ::= ¢ | <PARALLEL STATEMENT LIST>

<PARALLEL STATEMENT LIST> ::= <PARALLEL STATEMENT> |
                              <PARALLEL STATEMENT> <PARALLEL STATEMENT LIST>
```

2.11 WRITING AND RUNNING A SPINDLE PROGRAM

The previous sections described the components of a SPINDLE program. This section shows how a program is put together and how it runs as a whole. The syntax of a SPINDLE program is:

```
<SPINDLE PROGRAM> ::= <SPECIAL CHARACTER DECLARATION>
                      <RESERVED WORD DECLARATION>
                      <ATTRIBUTE DESCRIPTION>
                      <S-TERMINALS>
                      <NONTERMINAL DESCRIPTION>
                      <START SYMBOL DECLARATION>
                      <PROCEDURE DESCRIPTION>
                      <PRODUCTION DESCRIPTION>
```

Given a string of the language, a parse tree is built from the syntactic part of the definition. In the tree, ambiguous nodes have more than one subtree sprouting from them; S-terminal nodes have the corresponding attribute with the proper value filled in; nonterminal nodes have undefined attributes that correspond to the attribute identifiers associated with the nonterminal. Each nonterminal node is associated with a set of parallel statements. For each parallel

statement a process is created and activated. The execution of a process may create and activate other processes. A process may be passivated by the existence of a certain condition (e.g an undefined value); it is reactivated if and when the condition disappears. A process runs until it either passivates or terminates. The computation ends when there are no more active processes in the system. A computation that ends with no passive processes is said to be well-formed. If a computation is well-formed the following are all true:

- all ambiguities have been resolved and each node sprouts at most one subtree;
- all inherited and synthesized attributes are defined.

If a computation is malformed a list of passive processes is printed, showing the cause and location of the passivation. Notice that errors, unresolved ambiguities and circularities will all result in passivated processes.

2.12 THE DEFINITION OF TURINGOL

TURINGOL is a simple language that describes Turing machine programs. It was introduced, in a slightly different version, in Knuth [Kn 68a]. The following example gives the flavor of the language: it is a program designed to add unity to the binary integer that appears just left of the initially scanned square:

```
TAPE ALPHABET IS BLANK; ONE; ZERO; POINT;
PRINT 'POINT';
GO TO CARRY;
TEST: IF THE TAPE SYMBOL IS 'ONE' THEN
      (PRINT 'ZERO';
      CARRY: MOVE LEFT ONE SQUARE; GO TO TEST);
PRINT 'ONE';
REALIGN: MOVE RIGHT ONE SQUARE;
IF THE TAPE SYMBOL IS 'ZERO' THEN GO TO REALIGN.
```

The SPINDLE program in APPENDIX 2 defines the language. Given a well-formed string of TURINGOL, it will print its translation in TL/I. TL/I was introduced in Knuth [Kn 71], and is a machine-like language consisting essentially of sequential instructions whose operation codes are PRINT, MOVE, IF, JUMP and STOP. For example, for the TURINGOL program shown above, the SPINDLE program would print:

```
( 1: PRINT, 4)
( 2: JUMP, 5)
( 3: IF, 2, 7)
( 4: PRINT, 3)
( 5: MOVE, LEFT)
( 6: JUMP, 3)
( 7: PRINT, 2)
( 8: MOVE, RIGHT)
( 9: IF, 3, 11)
(10: JUMP, 8)
(11: STOP)
```

The difference between this version of TURINGOL and Knuth's original proposal is that, due to the introduction of empty declarations and the existence of empty statements, this version is ambiguous. For instance there are two possible parsings for the program:

```
TAPE ALPHABET IS A;; PRINT 'A'.
```

The modification was introduced to show how the disambiguation statement works. Notice that all parsings give the same meaning; however, since only one can be the correct one the definition states that: if the last declaration is empty the parsing is ambiguous and incorrect; if the first statement is empty but the last declaration is not the parsing is ambiguous and correct; otherwise the parsing is not ambiguous. This is an arbitrary choice imposed by SPINDLE's restriction that only one of the subtrees of an ambiguous node can be correct. The attribute EMPTY registers the existence of an empty last declaration or first statement. The disambiguation decision is made in the production for P because of the way the attributes were chosen and not because P is the possibly ambiguous node. By using an inherited attribute the information about the declaration being empty could be passed down the tree and then the disambiguation decision could be taken at some other node.

The binding of labels to addresses deserves a closer examination since essentially the same technique is used in the definition of SIMULA in Chapter 4. The present scheme is different from the one used by Knuth. The object programm OBJPROG is a list of instructions and pseudo-instructions. A label generates a pseudo-instruction that is placed in front of the labelled instruction. The pseudo-instruction has a component TAG to which is assigned a unique integer, the label-value. This label-value stands for the label; references to the labelled instruction are handled by assigning the label-value to a LABEL component. After OBJPROG (P) is defined the procedure OUTPUT builds a table that associates each label with an address and substitutes in the component LABEL of an instruction the label-value by the corresponding address. It should be noted that the

building of the table MAP and the assignment of addresses tc the LABEL components could not be done in one pass without the use of the procedure PLACE with a parallel statement for procedure body.

A fact that should be noted is that the definition states that TURINGOL programs containing undeclared identifiers are malformed, since a process trying to access the identifier in ENV will never terminate; however no explicit error message is printed. This way of indicating malformed programs while not wrong is not good programming practice: semantic errors should be explicitly stated. In the TURINGOL definition this could be accomplished by adding to the productions P21, P22, P23, P24 and P31 the parallel statement

```
$/ IF NULLR (FIND (FIND (ENV(S), SP(SIGMA)]), SYMBOL)) THEN
            ERROR (SP (SIGMA), "HAS NOT BEEN DECLARED") /$
```

and to P32 the same statement but with LABEL in place of SYMBOL.

It should be also noted that the printed output is an aspect of the meaning, not the whole meaning of the program since only part of OBJPROG is printed. However, since it can be presumed that the output reflects the essential aspects of the meaning, it is convenient to define the meaning associated by a SPINDLE definition with a string, as the printed output resulting from inputting the string.

Finally, it should be noted that since the application of the functions APEND and JOINE change the values of attributes lower in the tree, the final decorated tree does not correspond to the definition; the values of the attributes are not as stated in the definition. This can be avoided by using the # operator to copy at every stage. However, since one is only interested in the attributes of P, there is no harm in altering the values of the attributes of the other nodes of the parse tree.

CHAPTER 3

MUTILATE

This chapter describes the FOLDS machine MUTILATE. It is essentially a terse description of the relevant aspects of the machine implementation; the general concepts involved were explained in the preceding chapters.

MUTILATE is composed of two independent parts: the first comprises the parser and lexical analyzer; the second the interpreter. The first part reads in a string S and, if S belongs to the defined language, outputs a set (PT(S)). The second part reads in (PT(S)) and, if S is well-formed and the definition is well-formed, selects a PT(S) from the set and produces DPT(S). The main reason for this two level design is the particular nature of Earley's parsing algorithm [Ea 68], which is used in the parser for the reasons explained in Chapter 1. In Earley's scheme, the parsing of a string S is paced by the elements E of the string; i.e, the parsing develops by scanning the string from left to right and for each E building all possible partial parsing trees up to E. The trees are built in an extremely compact fashion with no duplication of nodes; i.e., a subtree representing the parsing of a substring common to two or more parsings is shared by the trees representing the parsings. While the parsing usually proceeds in a top down fashion, the parsing of left recursions is bottom up. It is difficult to recognize, at midparsing, subtrees that belong to the final parsing tree. While the parsing

usually proceeds from top to bottom, the subtrees are built on the way up. The combination of these characteristics makes the filling in of the semantics, while the parsing is going on, quite complicated. Thus it was decided that the advantages gained by developing the syntax and semantics at the same time would be offset by the complexity of the mechanisms necessary to carry out the task; it was considered more profitable, in a first stage, to develop the two tasks separately. This facilitated the development of the mechanisms for decorating the parse tree which was the main job at hand. Perhaps now that the semantic mechanisms are well understood, a one level process could be developed; but the complications are much more substantial then one would guess at first.

3.1 LEXICAL ANALYZER AND PARSER

The parser in MUTILATE is a straightforward implementation of Fisher's [Fi 70] version of Earley's algorithm, modified to accept empty substrings; the modification is a simple extension of the original algorithm. A table is used to speed up the parsing; it relates to each nonterminal the set of all the terminals that can be "seen" from the nonterminal. A terminal is seen from a nonterminal if either the terminal can be the first one in a string derived from the nonterminal or if there is a string of the language in which an empty substring that is followed by the terminal is derived from the nonterminal.

For each element of the string scanned, the parser calls the lexical analyzer. As described in Chapter 1 the analyzer recognizes special characters, reserved words, ALGOL-like identifiers, integers, and strings enclosed in double quotes, using blanks as separators. It also skips comments (beginning with the reserved word COMMENT and finishing with a semicolon) and an identifier following the reserved word END. When called, the lexical analyzer returns a token that identifies the recognized element; if the element is an S-terminal, it also returns the value to be assigned to the attribute associated with the node in the tree.

The parse tree is constituted of nonterminal and S-terminal nodes, organized as a left linked binary tree [Kn 68b]. Terminal nodes are ignored because they have no semantic consequence. A nonterminal node is divided into the fields SON, BROTHER, AMBIGUOUS, PRODUCTION and SELECTOR. SON contains a pointer to its rightmost son (that is not a terminal). BROTHER contains a pointer to its left brother. If the node is ambiguous, AMBIGUOUS points to another version of the same node (with a different subtree sprouting from it). PRODUCTION contains the label of the production associated with the node. SELECTOR contains the nonterminal identifier that labels the node. An S-terminal node is divided into the fields BROTHER, VALUE and SELECTOR. BROTHER is the same as for nonterminal, VALUE contains the value to be assigned to the attribute associated with the S-terminal and SELECTOR contains the S-terminal identifier that labels the node.

As an example, appendix 2 shows a TURINGOL program (the one presented in 2.12, with an empty declaration inserted) and the parsing tree generated from it.

Notice that common subtrees belonging to alternative ambiguous parsings are represented by a unique subtree; i.e, in an ambiguous subtree a node may belong to more than one parsing.

3.2 INTERPRETER

The interpreter is a multiple stack machine with four types of storage: byte addressed, linked, table and string. The byte addressed memory contains the instructions, the format descriptors and the nonterminal descriptors (a list of the symbol table entries for the attributes associated with a nonterminal). The linked storage contains nodes, attributes, stacks, etc., and is managed by an underlying garbage collection mechanism. The table storage contains a symbol table; there is one entry for each identifier (nonterminal, S-terminal, attribute or format), S-identifier and string in the definition of the language. The table also contains the S-identifiers and strings recognized by the parser. Each entry consists of a pointer to the spelling of the title in string storage, plus information about the "kind" of the entry (either attribute, nonterminal, S-terminal, format, S-identifier or string). If the entry corresponds to a nonterminal or a format, it contains the address of the respective description in byte addressed storage; if it corresponds to an S-terminal, it contains the symbol table address of the attribute associated with it; if it corresponds to an attribute, it contains the type and undertype of the attribute. In MUTILATE, a title value is represented by the address of its symbol table entry.

A MUTILATE segment is a sequence of MUTILATE instructions occupying contiguous positions in byte addressed storage; the address of a segment is the address of its first instruction. Each segment in storage corresponds to a parallel statement, procedure or function in the SPINDLE definition. A process is a dynamic instance of a segment and it is associated with a stack and a node. To execute a process is to interpret the instructions of its segment, starting with the first one; the instructions operate on the associated stack and the attributes of the associated node and its direct descendants. A process is represented by an element of linked storage called a Process Status Word (PSW) divided into the fields HEAD, STACK, VERSION, ID, LOC, and LINK. HEAD and STACK contain pointers to the associated node and stack respectively. VERSION and ID are used for disambiguation purposes; VERSION contains an integer and ID a pointer. LOC contains the address of an instruction: either the address of the segment or the address of an instruction that caused the passivation of the process. LINK contains a pointer and is used to link PSW's together in various lists as described below.

The interpreter operates in a pseudo parallel fashion with exactly one of the active processes (called the current process) being executed at any time; the register CURRENT points to its PSW. The PSWs of the remaining active processes are organized as a stack, called the PROCESS stack; the register PROCESS points to the top element of the stack. When the current process terminates its PSW is discarded; when it passivates, its PSW is transferred somewhere else. When a process is first activated, a PSW is created with its segment's address in the LOC field. When a process is reactivated its PSW is transferred to the PROCESS stack; the PSW's LOC field contains the address of the instruction that caused the passivation.

When the current process terminates or passivates, the one whose PSW is at the top of the stack is made current; CURRENT points to the PSW, which is removed from PROCESS. The MUTILATE registers LOC, HEAD and A are loaded with the contents of LOC(CURRENT), HEAD(CURRENT), and STACK(CURRENT) respectively. The MUTILATE register is then loaded with a pointer to the second element of the stack (if any). The process is then executed using the stack referenced by A and the node (and its direct descendants) referenced by HEAD. When a process passivates, the interpreter immediately stores the contents of LOC and A in LOC(CURRENT) and STACK(CURRENT) respectively and removes its PSW from CURRENT. In MANAGEMENT mode, the interpreter will make another process current. While a process is being executed the interpreter is in EXECUTE mode. When a process is terminated, the PSW pointed to by CURRENT is discarded and the interpreter switches to MANAGEMENT mode.

An attribute is represented as an element of linked storage divided into the fields TYPE, UNDERTYPE, SELECTOR, UND, IND, VALUE and LINK. TYPE and UNDERTYPE contain respectively the type and undertype associated with the attribute. SELECTOR contains the selector: if it is an integer, its negative value is stored; if it is a title, the address of its symbol table entry is stored. UND is a bit; if its value is 1, the attribute is undefined. Associated with every undefined attribute is a linked list formed by the PSWs of the processes passivated trying to access its value. The list is organized as a stack (using the LINK fields of the PSWs) and is called the interrupt stack. In an undefined attribute, VALUE contains a pointer to the associated interrupt stack. If the value of the bit field IND is 1, the attribute is indirect and VALUE contains a

pointer to another attribute. An attribute cannot be both undefined and indirect, thus UND and IND cannot both be 1. If UND and IND are both 0, VALUE contains the value of the attribute. If the attribute is elementary the field contains a value of the proper type. If the attribute is composite, its components form a linear list (using the LINK field) and VALUE contains a pointer to the first element of the linear list. If the attribute is of type LIST the components are ordered according to their position in the list; i.e, given a list attribute A, the first element in the linear list formed by the components is CAR(A), the second CAR(CDR(A)) and so on. If the attribute is of type CONSTRUCT the linear list is ordered in ascending order of the values of the SELECTOR field of the components. As a consequence a component whose selector is an integer always precedes a component whose selector is a title; a component whose selector is N (a positive integer) always follows a component whose selector is N+k (where k is a positive integer), because -N and -(N+k) are actually stored.

The processes' stacks are formed by attributes and PSWs linked through the PSWs' LINK fields. The attributes in the stack are always defined, direct and have an undefined SELECTOR field. The presence of a PSW in the stack indicates, as will be seen in section 3.3, that the stack is associated with a process which is a dynamic instance of a procedure or a function.

Nodes are represented as an element of linked storage divided into the fields SON, LEFTB, VALUE, SELECTOR, S-TERM, SEMANTICS, AMB, AMBIGUOUS, ONCE, CORRECT and DISAMB. SON contains a pointer to the rightmost direct descendant of the node. LEFTB contains a pointer to the sibling to the left of the node in the tree. The attributes

belonging to the node are organized as the components of a construct, and VALUE points to the first attribute in the linear list. SELECTOR contains the symbol table address of the entry that corresponds to the nonterminal or S-terminal identifier that labels the node. SEMANTICS is the address of the segment associated with the node. Only one segment is associated with a node; if the SPINDLE definition specifies more than one parallel statement for a node, the compiler encloses them in a parallel statement which is then the one associated with the node. For example, if the semantic rules of a production are embodied in the explicit parallel statements PST_1 and PST_2 and the implicit parallel statement PST_3, the compiler will associate with the production the segment generated for the parallel statement $/ PST_1 ; PST_2 ; PST_3 /$. AMB is a bit and if its value is 1 the node is ambiguous. In this case the field AMBIGUOUS contains a pointer to another version of the ambiguous node; otherwise AMBIGUOUS points to the node's nearest ambiguous ancestor. If the value of the bit feld ONCE is 1 the node is ambiguous, and the subtree sprouting from it has been tested. If the value of the bit field CORRECT is 1, the node is ambiguous and the subtree has been tested and found correct. If the value of the bit field DISAMB is 1 the node is ambiguous but has been found to have only one correct subtree which is the one sprouting from the node.

Notice that an ambiguous node is represented by a set of nodes, chained through the AMBIGUOUS field. The node at the head of the chain is called the (main) ambiguous node; the others are called versions of the node. In particular, the second node in the chain is called the auxiliary node of the main one. Only the main ambiguous

node belongs to the tree in the sense that ancestors and siblings point to it and not to its versions.

The interpreter initiates a run by loading the definition generated by the compiler into the various storages and building the tree produced by the parser in the linked storage. At the same time, AMBTABLE is built in the table storage; it associates an integer value (initially zero) with each main ambiguous node. AMBTABLE is used to purge from PROCESS those PSWs created while testing a subtree of an ambiguous node, once the testing is complete. Contrary to what was stated in Chapter 1, initially the nonterminal tree nodes have no attributes attached to them; attributes are created "on demand", by the execution of instructions. The tree is traversed depth-first, left to right, using a function and stack called DEVELOP. The stack contains pointers to the nodes of the tree; initially the stack contains a pointer to the root node. When DEVELOP is called it returns as a value, the pointer at the top of the stack; it also removes the top elemnt of the stack and inserts pointers to the descendants of the node referenced by the removed pointer. A call to DEVELOP when the stack is empty ends the run. When the node referenced by the value returned by DEVELOP, called the developing node, is ambiguous, a register AAMB is set to point to the node; otherwise AAMB is not touched. Then, each of the direct descendents D of the developing node is examined: if UND(D) = 0, set AMBIGUOUS(D) ← AAMB; otherwise set VALUE(AMBIGUOUS(D)) ← AAMB. This establishes the ancestor line of ambiguous nodes; each node points to its nearest ambiguous ancestor. If the node is not itself ambiguous the linking is done through the AMBIGUOUS field; otherwise through the value field of its auxiliary node. Initially the value of AAMB is NIL.

DEVELOP is called whenever the PROCESS stack is empty; it returns a pointer to the developing node. A PSW is then created, (which is a dynamic instance of the segment associated with the node), inserted in PROCESS and the run goes on. In the PSW, HEAD points to the node, STACK is NIL, and LOC contains the address of the associated segment obtained from the SEMANTICS field of the node. If the node is ambiguous ID is set to point to it, otherwise it is set to the same value as the field AMBIGUOUS of the node. VERSION is set to the same value as the entry in AMBTABLE corresponding to the pointer in ID.

When a PSW gets to the top of PROCESS, its VERSION and ID field are examined. If the value in VERSION is less than the value in AMBTABLE corresponding to the value in ID, the PSW is removed from the stack and discarded.

In addition to the tables mentioned above, the interpreter maintains a table, INTABLE, whose entries point to undefined nodes and main ambiguous nodes for which DISAMB=0. At the end of a run, if INTABLE is not empty, its contents are printed for diagnostic purposes.

3.3 THE INSTRUCTION SET OF MUTILATE

This section describes the instruction set of MUTILATE, basically a "Polish postfix" code analogous to Burroughs computers. The instructions are grouped according to their functions and a brief description of each one is presented. The description of their

execution by the interpreter follows the lines used by Knuth [Kn 68a] to describe algorithms. The definition of TURINGOL in MUTILATE assembly language, shown in appendix 3, illustrates the use of the instructions.

In addition to the registers mentioned in the previous sections (A, B, LOC, HEAD, and STACK), MUTILATE possesses registers X, Y, Z, OPCODE, OP1, and OP2. Here X, Y and Z are general purpose registers, OPCODE contains the designation of the instruction being executed and OP1 and OP2 its operands (if any).

The description of the executions utilizes an auxiliary procedure and an auxiliary function. The procedure, called PASSIVATE, takes one argument, a pointer to an undefined attribute U; when executed the procedure passivates the current processs, inserts its PSW into the interrupt stack of U and switches the interpreter to MANAGEMENT mode. The function, called FINAL, takes one argument, a pointer P to an attribute I; its execution can be described by the algorithm:

1. If IND(P)=0 return P.
2. Set P ← LINK(P) and go to 1.

The function returns the final attribute of I.

For instructions that do not belong to the "control" group (see section 3.3.4), when the execution is completed the instruction's length is added to the register LOC; notice that an instruction that causes a passivation does not complete its execution. For all instructions, when the machine is in EXECUTE mode, the next instruction to be executed is the one whose address is in LOC.

Section 3.3.8 contains an index with the opcodes of MUTILATE crossreferenced to the section number that explain them.

3.3.1 CONSTRUCT MANIPULATION INSTRUCTIONS

3.3.1.1 PLA & GET (Place and Get)

OPERANDS- OP1 is either an attribute identifier or empty; OP2 is either a node designation or empty. If OP1 is empty so is OP2 but the reverse may not be true.

STACK- If OP2 is not empty the stack does not matter. If OP1 is not empty but OP2 is, A is a pointer to a construct. If both OP1 and OP2 are empty A is either a title or an integer and B is a pointer to a construct.

DESCRIPTION- The instructions look for attributes in either a node or a construct, create them if they are not present and leave a pointer in the stack to the looked for attribute. A PLA instruction looking for a component in an undefined attribute will create a new attribute; under the same circumstances a GET instruction would cause a passivation. A PLA instruction looking for an attribute in an ambiguous node causes a passivation while a GET does not. Under all other circumstances, the two instructions behave in exactly the sam way.

EXECUTION-

1. If OP2 is empty go to 5. Set X ← "pointer to the node designated by OP2". If AMB=1 and DISAMB=0 and OPCODE=PLA, passivate the current process and discard its PSW.

2. Insert an attribute in the stack. Set TYPE(A) ← POINTER.
3. Look for the attribute whose selector is OP1, among the attributes of the node X; if the attribute is there, set VALUE(A) to point to it and END.
4. Create an attribute with the type and undertype associated with OP1 and set Y to point to it; set UND(Y) ← 1, VALUE(A) ← Y; make the attribute Y part of the linked list formed by the other attributes of the node X; END.
5. If OP1 is empty, set X ← FINAL(VALUE(B)); otherwise set X ← FINAL(VALUE(A)). If UND(X)=1 and OPCODE=PLA transfer the interrupt stack of X to PROCESS.
6. If OP1 is empty, set Z ← VALUE(A) and remove A from the stack; otherwise set Z ← OP1.
7. Look for the attribute whose selector is Z, among the components of X: if the attribute is there, set VALUE(A) to point to it and END.
8. Create an attribute. If Z is an attribute identifier the attribute has the type and undertype associated with Z; otherwise the undertype of X determines the type and undertype. Set Y to point to the attribute, UND(Y) ← 1, VALUE(A) ← Y; make the attribute Y part of the linked list formed by the other components of the attribute X.
9. END.

3.3.1.2. PLAN (Place New)

OPERANDS- OP1, an attribute identifier.

DESCRIPTION- This instruction creates a new attribute and leaves a pointer to it at the top of the stack. It is used to implement SPINDLE's assignment expression,

EXECUTION-

1. Insert a new element in the stack; set TYPE(A) ← POINTER.
2. Create an attribute whose type and undertype are the ones associated with OP1 and set Y to point to the attribute. Set UND(Y) ← 1, VALUE(A) ← Y.
3. END.

3.3.1.3 GETN (Get Next)

OPERANDS - None.

STACK- A pointer to an attribute.

DESCRIPTION- The instruction returns a pointer to the attribute that follows the one initially pointed at.

EXECUTION-

1. Set VALUE(A) ← LINK(VALUE(A)).
2. END.

3.3.1.4 FIND (Find)

OPERANDS- OP1 is either empty or is an attribute identifier.

STACK- If OP1 is empty A contains either a title or an integer and B is a pointer; otherwise A is a pointer.

DESCRIPTION- The instruction looks in a linked list for an attribute whose selector is given and leaves at the top of stack a pointer to it; if the linked list is empty or the attribute is not there, a NULL pointer is left at the top of the stack. This instruction is used to implement SPINDLE's function FIND.

EXECUTION-

1. If OP1 is empty, set Z ← VALUE(A) and remove A from the stack; otherwise set Z ← OP1.
2. If VALUE(A)=NIL, END.
3. Set X ← VALUE(A).
4. If SELECTOR(X)=Z, set VALUE(A) ← X and END.
5. If SELECTOR(X)>Z, set X ← LINK(X) and go to 4; otherwise set VALUE(A) ← NIL.
6. END.

3.3.1.5 FMT (Format)

OPERANDS- OP1, a format identifier.

STACK- A is a pointer to a construct.

DESCRIPTION AND EXECUTION- The instruction places a component FORMAT in a construct and assigns OP1 to it. The instruction is used to implement SPINDLE's format assignment. Its execution is equivalent to the execution of the sequence of MUTILATE instructions (PLA(FORMAT), ASSI(1 OP1)).

3.3.1.6 REP (Reproduce)

OPERANDS- None.

STACK- A is any attribute and B is a construct.

DESCRIPTION AND EXECUTION- The instruction implements the "*" operator of SPINDLE. The execution is equivalent to the execution of the sequence of MUTILATE instructions (NAME, PLA).

3.3.2 LIST MANIPULATION INSTRUCTIONS

The auxiliary procedure FIXLIST is used to describe the execution of list manipulation instructions. Its specification is:

ARGUMENTS- R, a register, either the A or B register.

DESCRIPTION- The procedure checks the attribute to which the register points. If it is a list attribute nothing happens. If it is a pointer to a list attribute then the pointer is substituted by

a list with the same components as the list attribute referenced by R.

EXECUTION-

1. If TYPE(R) = LIST, END.
2. Set X ← VALUE(R). If UND(X)=1, PASSIVATE(X).
3. Set TYPE(R) ← LIST, VALUE(R) ← VALUE(X).
4. RETURN.

3.3.2.1 CAR (Car)

OPERANDS- None.

STACK- A is either a list attribute or a pointer to one.

DESCRIPTION- A pointer to the first component of the list is left in the stack.

EXECUTION-

1. Execute FIXLIST(A). If VALUE(A)=NIL this is an error.
2. Set TYPE(A) ← POINTER.
3. END.

3.3.2.2 CDR (Cdr)

OPERANDS- None.

STACK- A is either a list attribute or a pointer to one.

DESCRIPTION- A new list is left in the stack, composed of all elements of the initial list but the first.

EXECUTION-

1. Execute FIXLIST(A). If VALUE(A)=NIL this is an error.
2. Set VALUE(A) ← LINK(VALUE(A)).
3. END.

3.3.2.3 CONS (Cons)

OPERANDS- None.

STACK- A is any attribute; B is either a list attribute or a pointer to one.

DESCRIPTION- The instruction inserts a new element at the front of the list in B.

EXECUTION-

1. Execute FIXLIST(B). Set X ← A. Remove A from the stack.
2. If TYPE(X)≠POINTER or VALUE(X)=NIL, set LINK(X) ← VALUE(A), VALUE(A) ← X and END.
3. Set X ← FINAL(VALUE(X)). Make a copy of the attribute referenced by X and place a pointer to the copy in Y. Set IND(Y) ← 1, VALUE(Y) ← X, LINK(Y) ← VALUE(A), VALUE(A) ← Y.
4. END.

3.3.2.4 LIST (List)

OPERANDS- None.

DESCRIPTION- A null list is inserted in the stack.

EXECUTION-

1. Insert a new attribute in the stack.
2. Set TYPE(A) ← LIST, VALUE(A) ← NIL.
3. END.

3.3.2.5 APEND (Append)

OPERANDS- None.

STACK- A is either a list attribute or a pointer to one; and so is B.

DESCRIPTION- The components of the list in A are appended to the list attribute in B by changing the link of the last component of B.

EXECUTION-

1. Execute FIXLIST(A), FIXLIST(B).
2. If VALUE(A)≠NIL, go to 5. If VALUE(B)≠NIL, set VALUE(B) ← VALUE(A), go to 5. Set X ← VALUE(B).
3. If LINK(X)≠NIL, set LINK(X) ← VALUE(A), go to 5.
4. Set X ← LINK(X), go to 3.
5. Remove A from stack.
6. END.

3.3.2.6 RVRS (Reverse)

OPERANDS- None.

STACK- A is a list attribute or a pointer to one.

DESCRIPTION- The instruction reverses the order of the components of the list.

EXECUTION-

1. Execute FIXLIST(A). If VALUE(A)=NIL or LINK(VALUE(A))=NIL, END; otherwise set X ← VALUE(A), Z ← NIL.
2. Set Y ← LINK(X), LINK(X) ← Z, Z ← X, X ← Y.
3. If X≠NIL go to 2. Set VALUE(A) ← Z.
4. END.

3.3.3 STACK MANIPULATION INSTRUCTIONS

3.3.3.1 POP (Pop)

OPERANDS- None.

STACK- A is any attribute.

DESCRIPTION AND EXECUTION- Remove the top element from the stack.

3.3.3.2 DBL (Double)

OPERANDS- None.

STACK- A is any attribute.

DESCRIPTION AND EXECUTIOM- A copy of the top element of the stack is inserted in the stack.

3.3.3.3 FLIP (Flip)

OPERANDS- None.

STACK- A and B are any attributes.

DESCRIPTION- The two top elements of the stack are interchanged.

EXECUTION-

1. Set X ← LINK(B), LINK(B) ← A, LINK(A) ← X, A ← B, B ← LINK(A).
2. END.

3.3.4 CONTROL INSTRUCTIONS

3.3.4.1 JUMP (Jump)

OPERANDS- OP1, the address of an instruction.

DESCRIPTION- Transfers control to the instruction whose address is OP1.

EXECUTION-

1. Set LOC ← OP1.
2. END.

3.3.4.2 JUMPF & JUMPT (Jump False and Jump True)

OPERANDS- OP1, the address of an instruction.

STACK- A is any attribute.

DESCRIPTION- Transfers control to the instruction whose address is OP1 if A contains the proper value. (TRUE if JUMPT or FALSE if JUMPF).

EXECUTION-

1. If VALUE(A)=FALSE or VALUE(A)=0 or VALUE(A)=NIL, set X ← FALSE; otherwise set X ← TRUE.
2. If X=TRUE and OPCODE=JUMPT, set LOC ← OP1 and END.
3. If X=FALSE and OPCODE=JUMPF, set LOC ← OP1 and END.
4. Set LOC ← LOC + L (where L is the length of the instruction).
5. END.

3.3.4.3 PAR & PARN (Parallel and Parallel New)

OPERAND- OP1 is the address of an instruction.

STACK- If OPCODE=PAR, A is any attribute.

DESCRIPTION- These instructions create a new PSW and insert it in PROCESS. PARN associates an empty stack with the new process; PAR associates a stack containing a copy of the top element of the current stack.

EXECUTION

1. Create a new PSW with a pointer to it in X.
2. Set LOC(X) ← OP1, HEAD(X) ← HEAD(CURRENT), ID(X) ← ID(CURRENT), VERSION(X) ← VERSION(CURRENT).
3. If OPCODE=PARN, set STACK(X) ← NIL; otherwise create a copy of the attribute in A and associate this one element stack with the new PSW.
4. Insert the new PSW in PROCESS.
5. END.

COMMENTS- PAR is used to implement parallel compound assignments. OP1 is the address of a segment.

3.3.4.4 CALL (Call)

OPERANDS- OP1, an instruction address.

DESCRIPTION- This instruction effects a procedure call. It passivates the current process and creates and activates a new one associated with the procedure.

EXECUTION-

1. Set LOC ← LOC + L (where L is the length of the instruction).
2. Create a new PSW, with a pointer to it placed in X. Set ID(X) ← ID(CURRENT), VERSION(X) ← VERSION(CURRENT), LOC(X) ← OP1. Create a new node and place a pointer to it in HEAD(X). Associate the current stack with the new PSW.
3. Passivate the current process and insert its PSW in the stack of the new process.
4. Make the new process current.
5. END.

3.3.4.5 RET (Return)

OPERANDS- None.

STACK- A or B is a PSW.

DESCRIPTION- This instruction returns control to the process that invoked the procedure.

EXECUTION-

1. If A is not a PSW, execute the MUTILATE instruction FLIP.
2. Remove the top element of A (a PSW), and leave a pointer to it in X. Set STACK(X) ← A.

3. Terminate the current process and make the X PSW current.
4. END.

3.3.4.6 HLT (Halt)

OPERANDS- None.

DESCRIPTION AND EXECUTION- The current process terminates, its PSW is removed from CURRENT and discarded. If the error condition is set the run terminates, otherwise the interpreter enters MANAGEMENT mode.

3.3.4.7 ERROR (Error)

OPERNADS- None.

DESCRIPTION AND EXECUTION- The error condition is set. As a consequence the first execution of a HLT instruction will terminate the run. Also any output instruction executed after this one, will never cause a passivation.

3.3.5 VALUE MANIPULATION INSTRUCTIONS

3.3.5.1 ASS (Assign)

OPERANDS- None.

STACK- A is any attribute; B is a pointer.

DESCRIPTION- This is the assignment instruction with the form B :≡ A.

EXECUTION

1. Set X ← VALUE(B). If UND(X)≠1, save the interrupt stack of X. Set UND(X) ← 0, IND(X) ← 0.
2. If TYPE(A)≠POINTER, set VALUE(X) ← VALUE(A) (TYPE(X)≠TYPE(A) must be true) and go to 6.
3. If VALUE(A)≠NIL, set VALUE(X) ← NIL (TYPE(X) must be either CONSTRUCT, LIST or POINTER) and go to 6.
4. If TYPE(X)≠POINTER, set VALUE(X) ← VALUE(A) and go to 6.
5. Set Y ← FINAL(VALUE(A)), IND(X) ← 1, VALUE(X) ← VALUE(Y). If UND(Y)≠1, insert the interrupt stack saved in step 1 (if any) into the interrupt stack of Y and go to 7.
6. Insert the interrupt stack saved in step 1 (if any) into the PROCESS stack.
7. Remove A and B.
8. END.

3.3.5.2 TRANS (Transfer)

OPERANDS- OP1 OP2 ... OPN, N≥2. OP1 contains the total number of operands, N, of the instruction; each of the following operands is a triple of the form (AT, NT_1, NT_2), where AT is an attribute identifier and the NTs node designations.

DESCRIPTION AND EXECUTION- Triples are executed in succession from left to right; the execution of each triple corresponds to the execution of the sequence of MUTILATE instructions (PLA(AT, NT_1), GET(AT, NT_2), ASS). The execution of a triple where NT_1 is an ambiguous node with DISAMB=0, passivates the current process and its PSW is discarded.

COMMENTS- TRANS is used to implement the generation of implied semantic rules. For ambiguous nodes, those triples which refer to inherited attributes should precede those that refer to synthesized attributes to guarantee that the inherited ones get assigned.

3.3.5.3 VALC (Value of a Constant)

OPERANDS- OP1, a constant.

DESCRIPTION- An attribute with value OP1 is inserted at the top of the stack.

EXECUTION

1. Insert an attribute in the stack whose type is the same as the type of OP1.
2. Set VALUE(A) ← OP1.
3. END.

3.3.5.4 ASSI (Assign Immediate)

OPERAND- OP1, a constant.

STACK- A points to an attribute.

DESCRIPTION- OP1 is assigned to the attribute referenced by VALUE(A).

EXECUTION

1. Execute the sequence of MUTILATE instructions (VALC(OP1), ASS).
2. END.

3.3.5.5 VAL (Value)

OPERANDS- OP1 is either empty or [.

STACK- A is any attribute.

DESCRIPTION- This instruction with operand "[" implements the bracketing operator of SPINDLE; with no operands it is used to implement the SPINDLE function FIRST, in conjuction with TEST

to implement the function NULLB, and in conjuction with COMP to implement the relation operator. Notice that unless A points to a composite attribute the operand is irrelevant.

EXECUTION-

1. If TYPE(A)≠POINTER or VALUE(A)=NIL, END.
2. Set X ← FINAL(VALUE(A)). If UND(X)=1, PASSIVATE(X).
3. If TYPE(X)=CONSTRUCT or TYPE(X)=LIST, if OP1≠[, set TYPE(A) ← POINTER, go to 4; otherwise END.
4. Set VALUE(A) ← VALUE(X).
5. END.

3.3.5.6 STO (Store)

OPERANDS- OP1, an attribute identifier.

STACK- A is any attribute.

DESCRIPTION- The instruction assigns the value in A to the local attribute whose selector is OP1. If the attribute is not found, one is created.

EXECUTION-

1. Execute the sequence of MUTILATE instructions (PLA(OP1,LOCAL), FLIP, ASS).
2. END.

3.3.5.7 LOAD (Load)

OPERANDS- OP1, an attribute identifier.

DESCRIPTION AND EXECUTION- The instruction "loads" the local attribute OP1 into the stack. If necessary an attribute is created.

EXECUTION-

1. Execute the sequence of MUTILATE instructions (GET(OP1,LOCAL), VAL).
2. END.

3.3.5.8 AR (Arithmetic)

OPERANDS- OP1, either ABS, NEG, ONEP (i.e. 1+), ONEM (i.e. -1+), +, -, *, /, REM.

STACK- A is an integer; B, if OP1 is a binary operator, is an integer.

DESCRIPTION- This is the arithmetic instruction. It performs the operation specified by OP1.

EXECUTION-

1. If OP1 is either ABS, NEG, ONEP or ONEM, set VALUE(A) ← OP1 VALUE(A) and END.
2. Set VALUE(B) ← VALUE(B) OP1 VALUE(A); remove A.
3. END.

3.3.5.9 LOG (Logical)

OPERANDS- OP1 is either NEG, AND or OR.

STACK- A is a boolean; B, if OP1 is either AND or OR, is a boolean.

DESCRIPTION- This instruction performs the logical operation specified by OP1.

EXECUTION-

1. If OP1=NEG, set VALUE(A) ← NEG VALUE(A) and END.
2. Set VALUE(B) ← VALUE(B) OP1 VALUE(A); remove A.
3. END.

3.3.5.10 TEST (Test)

OPERANDS- None.

STACK- A is any attribute.

DESCRIPTION- This instruction implements the SPINDLE functions NULLR and NULLB.

EXECUTION-

1. Set TYPE(A) ← BOOLEAN. If VALUE(A)=0 or VALUE(A)=FALSE or VALUE(A)=NIL, set VALUE(A) ← TRUE; otherwise set VALUE(A) ← FALSE.
2. END.

3.3.5.11 COMP (Compare)

OPERANDS- OP1, either =, ≠, >, ≥, ≤, <.

STACK- A and B are any attributes.

DESCRIPTION- This instruction implements SPINDLE's relations.

EXECUTION-

1. If the relation VALUE(B) OP1 VALUE(A) is TRUE, set VALUE(B) ← TRUE; otherwise set VALUE(B) ← FALSE. Set TYPE(B) ← BOOLEAN. Remove A.
2. END.

COMMENTS- TYPE(A) must be the same as TYPE(B).

3.3.5.12 NAME (Name)

OPERANDS- None.

STACK- A is a non-NIL pointer to an attribute whose selector is defined.

DESCRIPTION- This instruction implements the SPINDLE function SELECTOR.

EXECUTION-

1. Set X ← SELECTOR(VALUE(A)).
2. If X<0, set VALUE(A) ← -X, TYPE(A) ← INTEGER; otherwise set VALUE(A) ← X, TYPE(A) ← TITLE.

3. END.

3.3.5.13 GEN (Generate Numeric)

OPERANDS- None.

DESCRIPTION- Each time a GEN instruction is executed a unique integer is generated and placed in the stack.

EXECUTION-

1. Insert an attribute in the stack. Set TYPE(A) ← INTEGER, VALUE(A) ← new generated value.
2. END.

3.3.5.14 COPY (Copy)

OPERANDS- None.

STACK- A and B point to attributes with the same type and undertype.

DESCRIPTION- This instruction implements SPINDLE's copy operator.

EXECUTION-

1. Set Y ← FINAL(VALUE(A)). If UND(Y)=1, PASSIVATE(Y).
2. Set X ← VALUE(B). If UND(X)=1, set UND(X) ← 0 and transfer the interrupt stack of X to PROCESS. Set IND(X) ← 0. Remove A and B.
3. If TYPE(X)≠CONSTRUCT and TYPE(X)≠LIST, set VALUE(X) ← VALUE(Y) and END.

4. Set VALUE(X) ← NIL. For each component of Y an identical component is created, placed in X and the component of Y is indirectly assigned to the component of X.
5. END.

3.3.6 OUTPUT INSTRUCTIONS

MUTILATE maintains an output queue (OUTPUT) which is printed only when the run ends; this guarantees that for well-formed input strings, no undefined attributes are printed. The queue is composed of attributes placed in the queue by the output instructions. Each element of the queue corresponds to an output element of SPINDLE. The printing of each type of value and attribute was described in section 2.9.1. If UND=1 for an attribute of the printing queue, the current line is printed and a new one started; such an attribute corresponds to the output element "/". If IND=1 for an attribute of the printing queue, the output for this element is unformatted; i.e, the FORMAT component is considered as just another component. This is used for tracing purposes.

3.3.6.1 OUT & OUTF (Output and Output with Format)

OPERANDS- None.

STACK- A is any attribute.

DESCRIPTION- A is removed from the stack and placed in the printing queue.

EXECUTION

1. If OPCODE=OUT and the error condition is not set and there is an ambiguous node in the ancestor line of the current process node for which DISAMB=0, passivate the current process, discard its PSW and enter MANAGEMENT mode.
2. If OPCODE=OUT and if TYPE(A)=POINTER or TYPE(A)=LIST, set IND(A) ← 1.
3. Remove A and place it in the printing queue.
4. END.

3.3.6.2 OUTC (Output Control)

OPERANDS- None.

DESCRIPTION- The instruction puts a "/" operator in the printing queue.

EXECUTION-

1. If the error condition is not set and there is an ambiguous node in the ancestor line of the current process node for which DISAMB=0, passivate the current process, discard its PSW and enter MANAGEMENT mode.
2. Insert an attribute in the stack, set TYPE(B) ← POINTER and UND(A) ← 1.

3. Remove A and place it in the printing queue.

4. END.

3.3.7 THE DISAMBIGUATION INSTRUCTION - DAMB

OPERANDS- OP1 is either a nonterminal identifier or an integer.

STACK- A is a boolean attribute.

DESCRIPTION- This instruction implements the instruction DAMB of SPINDLE.

EXECUTION-

1. Set X ← ID(CURRENT). (ID points to the nearest ambiguous node in the ancestor line of the process' node.) If OP1 is an integer, set Z ← 1 and go to 4.
2. If SELECTOR(X)=OP1 go to 6.
3. Set X ← VALUE(AMBIGUOUS(X)) (get the nearest ambiguous ancestor of the node which is in the value field of its auxiliary node); go to 2.
4. If OP1=Z, go to 6.
5. Set Z ← Z +1, X ← VALUE(AMBIGUOUS(X)) and go to 4.
6. Set Z ← VALUE(A); remove A from the stack; if DISAMB(X)=1, END. (If the node is already disambiguated the instruction has no effect.)
7. Increase by 1 the value corresponding to X in AMBTABLE. Eliminate the LOCAL attributes of X.
8. Go through the subtree originating from X and for all developed nodes that are not S-terminals set the VALUE

field to NIL. For those nodes that are not ambiguous set AMBIGUOUS to NIL. For an ambiguous node increase the corresponding entry in AMBTABLE by 1; set the bits ONCE and CORRECT in all the versions of the node to 0; set VALUE of its auxiliary node to NIL. (The tree must be cleared because one node may belong to more than one ambiguous subtree.) Eliminate from the DEVELOP stack any element that points to one of the nodes of the subtree.

9. If Z=FALSE, set CORRECT(X) ← 0; otherwise set CORRECT(X) ← 1. (If Z=TRUE and one of the versions of X has CORRECT=1, an error occurs.)
10. Go through all the versions of X and look for one whose bit ONCE has value 0. If none is found go to 12; otherwise set Y to point to the one found.
11. (Another parsing is tested.) Set CORRECT(Y) ← CORRECT(X), ONCE(Y) ← 1, Z ← SON(X), SON(X) ← SON(Y), SON(Y) ← Z, and go to 14.
12. (All parsings have been tried.) If CORRECT(X)=0 and for all versions V of X, CORRECT(V)=0, set X ← VALUE(AMBIGUOUS(X)) and go to 7. (All parsings are incorrect so try the nearest ambiguous node in the ancestor line.)
13. (There is one correct parsing.) Set DISAMB(X)=1. If CORRECT(X)≠1, set Y to point to the version for which CORRECT is 1, Z ← SON(X), SON(X) ← SON(Y), SON(Y) ← Z.
14. Insert a pointer to X in DEVELOP. Passivate the current process and discard its PSW. Enter MANAGEMENT mode.
15. END.

COMMENT- The existing implementation of MUTILATE has a different DAMB than the one described here. As implemented now, once a correct parsing is found, the other parsings are not tested; the PSWs corresponding to the processes interrupted trying to "PLA" the synthesized attributes of the node (that are saved instead of being discarded) are inserted in PROCESS, DISAMB is set to 1 and the current process continues.

3.3.8 INDEX OF OPCODES

OPCODE	SECTION
APEND	3.3.2.5
AR	3.3.5.8
ASS	3.3.5.1
ASSI	3.3.5.4
CALL	3.3.4.4
CAR	3.3.2.1
CDR	3.3.2.2
COMP	3.3.5.11
CONS	3.3.2.3
COPY	3.3.5.14
DAMB	3.3.7
DBL	3.3.3.2
ERROR	3.3.4.7
FIND	3.3.1.4
FLIP	3.3.3.3
FNT	3.3.1.5
GEN	3.3.5.13
GET	3.3.1.1
GETN	3.3.1.3
HLT	3.3.4.6
JUMP	3.3.4.1
JUMPF	3.3.4.2
JUMPT	3.3.4.2
LIST	3.3.2.4
LOAD	3.3.5.7
LOG	3.3.5.9
NAME	3.3.5.12
OUT	3.3.6.1
OUTC	3.3.6.2
OUTF	3.3.6.1
PAR	3.3.4.3

PARN	3.3.4.3
PLA	3.3.1.1
PLAN	3.3.1.2
POP	3.3.3.1
REP	3.3.1.6
RET	3.3.4.5
RVRS	3.3.2.6
STO	3.3.5.6
TEST	3.3.5.10
TRANS	3.3.5.2
VAL	3.3.5.5
VALC	3.3.5.3

CHAPTER 4

A DEFINITION OF SIMULA

This chapter contains the SPINDLE defintion of a subset of the SIMULA 67 Common Base Language [DMN 70] . The definition is closely patterned after Wilner's definition of SIMULA [Wi 71]; it is intended to show the viability of FOLDS for the definition of large programming languages. The definition also serves as an example of a variety of SPINDLE features and programming techniques.

The definition is essentially an implementation of Wilner's definitions. Modifications were introduced mainly where errors were found and where they simplified the definition without changing its character. Whenever possible, the attributes' names and structures were processed as in Wilner's specification. However, the present definition does differ from Wilner's in three important aspects. First, the present definition takes into account the existence of SPINDLE's lexical analyzer. Second, labels are handled here as in TURINGOL, contrary to the technique used by Wilner which resembles Knuth's technique in TURINGOL; the implementation of Wilner's scheme in SPINDLE would be very costly in terms of the number of semantic rules necessary to process the two attributes he called α and Z. The third difference is in the way the target language program is handled. In this definition, a program is a set of pairs, each consisting of a segment and its designation; each segment stands for a sequence of instructions. Wilner uses an attribute R which

collects such pairs throughout the tree and carries them to the root node. In the present definition instead of collecting the segments, they are printed by the function OUTPUT at each node where they occur. This simplifies the definition by doing away with the attribute R which would otherwise occur throughout the tree. The code generated from the present definition runs in the machine defined by Wilner [Wi 71] modified as follows:

- The instruction CHE has an additional field CLASS, containing a boolean value.
- The instruction MAK has an additional field COPIES, containing an integer value.
- The instruction GEN, after creating the new object and transferring the actual $_{w}$'s to its stack, creates a new stack level by placing a ret $_{w}$ and a mark $_{w}$ in the stack.
- The instruction CHE in a first step, if the CLASS field is TRUE, copies to the top of the stack the actual $_{w}$ which is in the next lower level in the stack and whose stack displacement is given by the field D of the instruction.
- Step 3 in the execution of both VAL and ADDR is modified so that the address left in the stack is not a pointer to the lowest mark $_{w}$ in the stack of the remote object but to the one above the lowest.
- The last step in the execution of MAK is modified so that before "fin", a number of array s $_{w}$ (equal to the value in the

COPIES field), are placed in the stack. For each new array $_{w}$ a copy of the structure pointed to by the array $_{w}$ at the top of the stack is created with the new array $_{w}$ pointing to the new structure.

These modifications are necessitated due to changes made in Wilner's definition to correct the mechanism for concatenating class segments and to correct the mechanism for creating arrays declared in the same array segment.

This chapter has two sections: the first contains the definiton of SIMULA in SPINDLE, the second a comparison of the definition with Wilner's definition. Appendix 4 contains a set of SIMULA programs and the target language generated from them by the definition running in MUTILATE.

4.1 DEFINITION

```
TERMINALS ARE + - * / ( ) [ ] . , ¬ < = > : ;

RESERVED WORDS ARE AND, ARRAY, BEGIN, BOOLEAN, CLASS, DETACH, DIV,
                   DO, ELSE, END, EQUIV, FALSE, GO, IF, IMPLIES, IN,
                   INNER, INSPECT, INTEGER, IS, LABEL, NAME, NEW,
                   NONE, OR, OTHERWISE, PROCEDURE, QUA, REAL, REF,
                   RESUME, SWITCH, THEN, THIS, TO, TRUE, VALUE,
                   VIRTUAL, WHEN, WHILE

ATTRIBUTES ARE

   ADDR = CONSTRUCT
   AEMDEC = BOOLEAN
```

```
ALSO = BOOLEAN
APA = INTEGER
ARULE = RULE
ATTR = INTEGER
BEGUN = BOOLEAN
CDECL = LIST
CL = INTEGER
CLASSN = POINTER
CODE = RULE
COND = BOOLEAN
D0 = INTEGER
D = D0
DAR = BOOLEAN
DISP = INTEGER
DN = INTEGER
E = CONSTRUCT, PL
E1 = E
EMDEC = BOOLEAN
ENV = E
ENV1 = ENV
ENVA = ENV
FIRSTST = BOOLEAN
FJUMP = LABEL1
FORMALE = E
GENUS = CONSTRUCT
GENUS1 = GENUS
GENUS2 = GENUS
INSTR = CONSTRUCT
ITEM = LIST
JLABEL = INTEGER
KIND = TITLE
L = INTEGER
LABEL1 = INTEGER
LEGIT = INTEGER
LEVEL = INTEGER
LL = INTEGER
LN = INTEGER
LOCALE = E
MAP = CONSTRUCT, INTEGER
MARK = TITLE
MARK1 = TITLE
MAT = CONSTRUCT, MATVEC
MATRIX = MAT
MATVEC = CONSTRUCT
MOAMB = BOOLEAN
MODE = TITLE
N = INTEGER
N1 = N
N2 = N
NAMETB = NTB
NEXT1 = POINTER
NEXT2 = NEXT1
NEXT3 = NEXT1
NFORMALS = INTEGER
NLOCALS = INTEGER
NOLABEL = BOOLEAN
NTB = CONSTRUCT, INTEGER
NUMDEC = INTEGER
```

```
O = OPEN
OBJECT = INTEGER
OPEN = TITLE
OPER = TITLE
ORIG = INTEGER
OUTERMOST = BOOLEAN
PL = CONSTRUCT
PL1 = PL
PLACE = TITLE
PPL = PL
PREF = INTEGER
PREFIX = POINTER
QTBVEC = CONSTRUCT
QUAL = INTEGER
QUAL1 = QUAL
QUAL2 = QUAL
QUALTB = CONSTRUCT, QTBVEC
RULE = LIST
RULE1 = RULE
RULE2 = RULE
SEGMENT = INTEGER
SID = BOOLEAN
SL = SN
SM = SN
SN = INTEGER
SP = TITLE
SPEC = GENUS
START = BOOLEAN
T = TITLE
TYPD = GENUS
TYPDS = TYPD
TYPE = TITLE
TYPE1 = TYPE
TYPE2 = TYPE
TJUMP = LABEL1
UNDECL = RULE
USE = TITLE
V = INTEGER
VALENCE = INTEGER
VIRDECL = CONSTRUCT, INSTR
VIRDECL1 = VIRDECL
VIRTUALE = E
XX = CONSTRUCT
```

COMMENT

THE ATTRIBUTES E AND ENV REPRESENT THE SYMBOL TABLE: E COLLECTS THE DECLARATIONS THAT ARE SPREAD THROUGH THE TREE BY ENV. EACH ENTRY

OF THE SYMBOL TABLE IS A CONSTRUCT OF TYPE PL (FOR PROPERTY LIST) WITH THE SELECTOR CONTAINING THE SPELLING, AND WITH THE COMPONENTS OF THE ENTRY REPRESENTING THE PROPERTIES OF THE IDENTIFIER. PL MAY HAVE COMPONENTS GENUS, ADDR, ATTR, N, NFORMALS, NLOCALS, LOCALE, FORMALE, VIRTUALE, SEGMENT, PREF, OBJECT AND CODE. GENUS CONTAINS THE TYPE AND KIND OF AN IDENTIFIER: WHEN THE TYPE IS "REF" IT ALSO CONTAINS A COMPONENT QUAL, WHOSE VALUE IS THE SEGMENT DESIGNATION OF THE CLASS THAT QUALIFIES THE REFERENCE. ADDR IS THE STACK ADDRESS OF THE INSTRUCTION CORRESPONDING TO THE DECLARATION OF THE IDENTIFIER: IT HAS COMPONENTS LN, THE STACK LEVEL, AND DN, THE STACK DEPTH. FOR IDENTIFIERS THAT ARE CLASS ATTRIBUTES (ATTRIBUTES HERE IN THE SIMULA SENSE) ATTR CONTAINS THE SEGMENT DESIGNATION OF THE CLASS, OTHERWISE IT CONTAINS A 0. N GIVES THE NUMBER OF DIMENSIONS ASSOCIATED WITH ARRAY IDENTIFIERS AND THE LENGTH OF THE SWITCH LIST ASSOCIATED WITH A SWITCH IDENTIFIER. NFORMALS GIVES THE NUMBER OF FORMAL PARAMETERS FOR CLASSES AND PROCEDURE IDENTIFIERS. THE REMAINING COMPONENTS OF PL ARE ASSOCIATED ONLY WITH CLASS IDENTIFIERS. NLOCALS CONTAINS THE NUMBER OF ATTRIBUTES (IN THE SIMULA SENSE) OF A CLASS. LOCALE IS A SYMBOL TABLE WHOSE ENTRIES ARE THE ATTRIBUTES OF THE CLASS. FORMALE IS THE SYMBOL TABLE FOR THE FORMAL PARAMETERS. VIRTUALE IS THE SYMBOL TABLE FOR THE VIRTUAL ATTRIBUTES OF THE CLASS IDENTIFIER. SEGMENT CONTAINS THE SEGMENT DESIGNATION OF THE CLASS WHICH IS THE DESIGNATION NUMBER OF THE SEGMENT ASSOCIATED WITH THE CLASS. PREF CONTAINS THE SEGMENT DESIGNATION OF THE PREFIX CLASS. OBJECT CONTAINS THE SEGMENT DESIGNATION OF THE PROTOTYPE ASSOCIATED WITH THE CLASS IDENTIFIER. CODE CONTAINS THE RULE THAT STANDS FOR THE SEGMENT ASSOCIATED WITH THE CLASS.

PL IS ALSO USED TO CONVEY THE PROPERTY LIST OF EXPRESSIONS AND

THEIR COMPONENTS. PPL IS USED TO PASS TO THE MAIN PART OF A CLASS DECLARATION THE PROPERTY LIST ASSOCIATED WITH THE PREFIX.

RULE STANDS FOR A SEQUENCE OF INSTRUCTIONS IN THE TARGET LANGUAGE: IT IS OF TYPE LIST WITH COMPONENTS OF TYPE INSTR. RULE CONTAINS BOTH TARGET LANGUAGE INSTRUCTIONS AND PSEUDO-INSTRUCTIONS: TARGET LANGUAGE INSTRUCTIONS HAVE A COMPONENT FORMAT, PSEUDO-INSTRUCTIONS DO NOT. A PSEUDO-INSTRUCTION WITH COMPONENT LABELI STANDS FOR A LABEL WITH LABELI CONTAINING THE UNIQUE INTEGER ASSOCIATED WITH THE LABEL. A COMPONENT MARK IDENTIFIES THE PSEUDO-INSTRUCTIONS THAT MARK THE POSITIONS OF "INIT" AND "INNER" IN THE SEGMENT ASSOCIATED WITH A CLASS. A COMPONENT MARK1 IDENTIFIES THE PSEUDO-INSTRUCTIONS THAT ENCLOSE THE SEQUENCE OF INSTRUCTIONS CORRESPONDING TO THE CALCULATION OF ARRAY BOUNDS. (SEE THE FUNCTION VIRMERGE FOR AN EXPLANATION OF THE USE OF THOSE MARKERS). INSTRUCTIONS THAT REFER TO LABELS CONTAIN A COMPONENT JLABEL WHOSE VALUE IS THE UNIQUE INTEGER ASSOCIATED WITH THE LABEL. THE PROCEDURE OUTPUT BINDS LABELS TO ADDRESSES AND ASSOCIATES THE ADDRESS ASSOCIATED WITH THE LABEL IN JLABEL WITH THE COMPONENT DISP OF THESE INSTRUCTIONS. THE COMPONENT OPER IS USED IN VARIOUS INSTRUCTIONS TO HOLD AN OPERAND FOR THE INSTRUCTIONS.

UNDECL IS A LIST OF THE SAME NATURE AS RULE AND IS USED TO COLLECT THE INSTRUCTIONS RESULTING FROM THE DECLARATION OF LABELS.

VIRDECL IS A CONSTRUCT WHOSE COMPONENTS ARE INSTRUCTIONS RESULTING FROM THE "REDECLARATION" OF VIRTUAL CLASS ATTRIBUTES. THESE INSTRUCTIONS REPLACE THE INSTRUCTIONS ASSOCIATED WITH THE PREVIOUS DECLARATIONS OF THE CLASS ATTRIBUTES: THE FUNCTION VIRMERGE REPLACES THE INSTRUCTIONS ASSOCIATED WITH THE PREVIOUS DECLARATION BY THE INSTRUCTIONS IN VIRDECL.

ENV1 IS USED TO PROPAGATE THE VIRTUALE OF A CLASS SO THAT THE REDEFINITION OF VIRTUAL IDENTIFIERS CAN BE PERFORMED.

ENVA IS USED TO CARRY THE OUTER ENVIRONMENT OF A PROCEDURE OR A CLASS (PLUS THE FORMAL PARAMETERS) TO THE EXPRESSION THAT CALCULATES THE BOUNDS OF AN ARRAY WHICH HAS BEEN DECLARED EITHER IN A PROCEDURE OR A CLASS BODY.

CL IS USED TO CONVEY THE SEGMENT DESIGNATION OF A CLASS TO THE DECLARATION OF ITS ATTRIBUTES.

DO AND D ARE USED TO CALCULATE THE STACK DISPLACEMENT CORRESPONDING TO AN IDENTIFIER DECLARATION. D CAN ALSO BE VIEWED AS THE NUMBER OF IDENTIFIERS DECLARED PRIOR TO THE IDENTIFIER DECLARATION.

LL INDICATES THE LEXICOGRAPHICAL LEVEL OF AN IDENTIFIER AND ALSO THE STACK LEVEL ASSOCIATED WITH IT.

L INDICATES THE LENGTH OF A LIST SUCH AS A FORMAL PARAMETER LIST.

OUTERMOST IS USED TO DISTINGUISH A STATEMENT WHICH IS A CLASS' OUTER BLOCK.

TYPD AND TYPDS ARE USED TO CONVEY GENUS IN A DECLARATION. TYPDS GETS THE GENUS FROM THE SPECIFIER AND TYPD TAKES IT TO THE VARIABLES IN THE DECLARATION.

USE CONVEYS THE USE OF AN EXPRESSION: FOR ITS VALUE, FOR ITS ADDRESS, OR FOR LATER EXECUTION (AS A PARAMETER CALLED BY NAME).

VALENCE CLASSIFIES "+" OR "-" AS EITHER UNARY OR BINARY.

NOLABEL IS USED TO AVOID RECOGNIZING A LABELLED BLOCK MORE THAN ONCE.

BEGUN IS USED TO IDENTIFY BLOCKS THAT ARE EITHER A CLASS OR PROCEDURE BODY.

FJUMP AND TJUMP ARE USED TO PASS THE LABEL OF AN INSTRUCTION IN A CONDITIONAL OR CONNECTION STATEMENT.

V AND SP ARE ATTRIBUTES ASSOCIATED WITH STRUCTURED TERMINALS: V CONTAINS THE VALUE OF AN INTEGER AND SP THE SPELLING OF AN IDENTIFIER.

ORIG CONTAINS THE SEGMENT DESIGNATION OF THE SEGMENT WHICH CONTAINS THE FIRST INSTRUCTION OF A SIMULA PROGRAM.

ALSO IS USED TO RECOGNIZE AN ASSIGNMENT STATEMENT WHICH IS ITSELF A RIGHT HAND SIDE OF AN ASSIGNMENT STATEMENT.

LEGIT SERVES TO INDICATE WHETHER A SPECIFICATION PART BELONGS TO A PROCEDURE OR A CLASS.

PLACE IDENTIFIES THE CONTEXT OF AN IDENTIFIER LIST: SPECIFICATION PART, NAME PART, VALUE PART OR VIRTUAL PART.

QUALTB IS A TABLE, IN WHICH EACH ENTRY CORRESPONDS TO A CLASS. EACH COMPONENT OF QUALTB IS A CONSTRUCT WHOSE SELECTOR IS THE SEGMENT DESIGNATION OF THE CLASS AND WHOSE COMPONENTS ARE PREFIX, CLASSN AND LEVEL. PREFIX CONTAINS A POINTER TO THE QUALTB ENTRY CORRESPONDING TO THE PREFIX CLASS. CLASSN CONTAINS A POINTER TO THE SYMBOL TABLE ENTRY CORRESPONDING TO THE CLASS. LEVEL CONTAINS THE NUMBER OF CLASSES IN THE PREFIX SEQUENCE OF THE CLASS.

CDECL IS A LIST OF POINTERS TO THE SYMBOL TABLE ENTRIES CORRESPONDING TO THE CLASSES DECLARED IN A BLOCK. IT IS USED BY THE FUNCTION UPDQUALTB TO CREATE NEW ENTRIES IN QUALTB.

NTB AND NAMETB ARE CONSTRUCTS THAT ESTABLISH THE CORRESPONDENCE BETWEEN FORMAL PARAMETERS AND THEIR POSITION IN THE STACK: THEIR COMPONENTS ARE INTEGERS WHOSE SELECTORS ARE THE SPELLING OF THE FORMAL PARAMETERS AND WHOSE VALUES ARE THEIR STACK DISPLACEMENT.

MAT AND MATRIX ARE CONSTRUCTS USED TO ESTABLISH THE

CORRESPONDENCE BETWEEN FORMAL PARAMETERS AND THEIR PROPERTIES. EACH COMPONENT IS A CONSTRUCT WHOSE SELECTOR IS THE STACK DISPLACEMENT OF THE FORMAL PARAMETER AND WHOSE COMPONENTS ARE MODE AND SPEC. MODE CONTAINS THE MODE OF TRANSMISSION OF THE PARAMETER AND SPEC ITS GENUS.

ITEM IS A LIST OF CONSTRUCTS, EACH CORRESPONDING TO A CLASS ASSOCIATED WITH AN ENCLOSING CONNECTION BLOCK. THE COMPONENTS XX OF ITEM CONTAIN COMPONENTS ADDR AND QUAL. QUAL CONTAINS THE SEGMENT DESIGNATION OF THE CLASS AND ADDR THE STACK DESIGNATION OF A REFERENCE TO THE CONNECTED OBJECT.

MAP IS USED, AS IN TURINGOL, TO BIND LABELS AND ADDRESSES.

SM, SN AND SL CONTAIN SEGMENT DESIGNATIONS OR THE UNIQUE INTEGERS THAT REPRESENT LABELS.

APA, COND, DAR, AND SID ARE USED FOR DISAMBIGUATION PURPOSES. THEY SERVE TO DETECT AND RESOLVE AMBIGUITIES ARISING FROM ACTUAL PARAMETERS AND LEFT HAND SIDE OF VALUE ASSIGNMENTS WHEN THEY PARSE TO A SINGLE ENTITY. A SINGLE ENTITY IS EITHER AN IDENTIFIER (POSSIBLY REMOTE), OR AN IDENTIFIER FOLLOWED BY AN EXPRESSION ENCLOSED IN SQUARE BRACKETS, OR A FUNCTION DESIGNATOR, OR A CONDITIONAL EXPRESSION WHOSE THEN AND ELSE PARTS ARE BOTH SINGLE ENTITIES, OR A SINGLE ENTITY ENCLOSED IN PARENTHESIS. THEY ALSO SERVE TO DETECT THE AMBIGUITY ARISING FROM A PRIMARY THAT PARSES TO AN IDENTIFIER.

FIRSTST, EMDEC, AEMDEC, AND NUMDEC ARE USED TO RESOLVE THE AMBIGUITIES ARISING FROM COMPOUND STATEMENTS WHERE THE FIRST STATEMENT IS EMPTY AND FROM UNLABELLED BLOCKS WHERE THE FIRST STATEMENT OF THE COMPOUND TAIL IS EMPTY. FIRSTST AND EMDEC IN CONJUCTION WITH START ARE USED TO DISAMBIGUATE INITIAL OPERATIONS WHOSE FIRST STATEMENT IS EMPTY.

OPEN AND O ARE USED TO RESOLVE THE AMBIGUITY ARISING FROM AN INSPECT STATEMENT WITH A MATCHING OTHERWISE CLAUSE WHICH IS INSIDE ANOTHER INSPECT STATEMENT WITHOUT A MATCHING OTHERWISE CLAUSE;

FOLLOWING IS A GLOSSARY OF THE ATTRIBUTE IDENTIFIERS USED IN THIS DEFINITION:

ADDR - STACK ADDRESS. USUALY A COMPONENT A SYMBOL TABLE ENTRY.

AEMDEC -USED FOR DISAMBIGUATION. TRUE IF ALL THE DECLARATIONS IN A BLOCK ARE EMPTY.

ALSO - DETECTS MULTIPLE LEFT-HAND SIDES IN AN ASSIGNMENT STATEMENT.

APA - USED FOR DISAMBIGUATION PURPOSES

ARULE - THE RULE GENERATED BY THE DECLARATION PART OF A SPLIT BODY.

ATTR - FOR CLASS ATTRIBUTES, THE SEGMENT DESIGNATION OF THE CLASS. A COMPONENT OF A SYMBOL TABLE ENTRY.

BEGUN - DETECTS A BLOCK AS A CLASS BODY, PROCEDURE BLOCK OR CONNECTION BLOCK.

CDECL - LIST OF POINTERS TO THE SYMBOL TABLE ENTRIES CORRESPONDING TO CLASS DECLARATIONS IN A BLOCK.

CL - CARRIES THE SEGMENT DESIGNATION OF A CLASS TO THE CLASS ATTRIBUTE'S DECLARATION.

CLASSN - POINTER TO A SYMBOL TABLE ENTRY FOR A CLASS. A COMPONENT OF A QUALTB ENTRY.

CODE - THE RULE ASSOCIATED WITH A CLASS. A COMPONENT OF A SYMBOL TABLE ENTRY FOR A CLASS IDENTIFIER.

COND - A PARAMETER FOR THE PROCEDURES DISAMV AND DISAMF.

D AND DO -USED TO CALCULATE THE STACK DISPLACEMNT ASSOCIATED WITH IDENTIFIERS.

DAR - USED FOR DISAMBIGUATION PURPOSES. TRUE IF A VARIABLE IS AN ARRAY ELEMENT, FALSE OTHERWISE.

DISP - DISPLACEMENT OF AN INSTRUCTION IN A SEGMENT. USUALLY A COMPONENT OF INSTR.

DN - STACK DISPLACEMENT OF A VARIABLE. USUALLY A COMPONENT OF ADDR.

E - COLLECTS SYMBOL TABLE ENTRIES.

EMDEC - USED FOR DISAMBIGUATION PURPOSES. IT IS TRUE IF THE LAST DECLARATION IN A BLOCK HEAD IS EMPTY.

ENV - THE SYMBOL TABLE: THE ENVIRONMENT.

ENV1 - A SYMBOL TABLE FOR VIRTUAL ATTRIBUTES.

ENVA - A SYMBOL TABLE FOR USE BY THE BOUNDS IN AN ARRAY DECLARATION.

FIRSTST - DETECTS AN EMPTY FIRST STATEMENT.

FJUMP - UNIQUE INTEGER THAT LABELS THE INSTRUCTION FOLLOWING THE INSTRUCTIONS TO BE SKIPPED IN A CONDITIONAL STATEMENT.

FORMALE - SYMBOL TABLE FORMED BY THE FORMAL PARAMETERS OF A CLASS. A COMPONENT OF A SYMBOL TABLE ENTRY.

GENUS - THE PROPERTIES OF AN IDENTIFIER: TYPE, KIND AND CLASS QUALIFICATION.

INSTR - AN INSTRUCTION OF THE TARGET LANGUAGE. USUALLY A COMPONENT OF RULE, UNDECL OR VIRDECL.

ITEM - LIST USED FOR REFERENCING OBJECTS ENCLOSING A CONNECTION BLOCK. COMPONENTS ARE CONSTRUCTS WITH COMPONENTS QUAL AND ADDR. QUAL IS THE QUALIFICATION OF THE OBJECT AND ADDR THE STACK ADDRESS OF A POINTER TO THE OBJECT.

JLABEL - THE UNIQUE INTEGER ASSOCIATED WITH THE LABEL OF AN INSTRUCTION. USUALLY A COMPONENT OF INSTR.

KIND - THE KIND OF AN IDENTIFIER (IN THE ALGOL SENSE).

L - LENGTH OF A LIST SUCH AS SUBSCRIPT LIST, PARAMETER LIST, AND ETC..

LABELI - UNIQUE INTEGER ASSOCIATED WITH A LABEL. A COMPONENT OF A PSEUDO-INSTRUCTION WHICH CORRESPONDS TO A LABEL.

LEGIT - MARKS A SPECIFICATION PART AS BELONGING TO A PROCEDURE, A CLASS HEADING OR A VIRTUAL PART.

LEVEL - THE PREFIX LEVEL OF A CLASS. A COMPONENT OF A QUALTB ENTRY.

LL - THE LEXICOGRAPHICAL LEVEL: THE STACK LEVEL.

LN - THE STACK LEVEL OF A VARIABLE. USUALLY A COMPONENT OF ADDR.

LOCALE - SYMBOL TABLE FORMED BY THE ATTRIBUTES OF A CLASS. A COMPONENT OF A SYMBOL TABLE ENTRY.

MAP - TABLE THAT RELATES THE UNIQUE INTEGERS REPRESENTING LABELS TO THE ACTUAL ADDRESSES ASSOCIATED WITH THE LABELS.

MARK - COMPONENT OF PSEUDO-INSTRUCTION MARKING THE LOCATION OF "INIT" OR "INNER" IN THE RULE CORRESPONDING TO A CLASS BODY.

MARK1 - COMPONENT OF A PSEUDO-INSTRUCTION MARKING THE BOUNDARIES OF THE BOUND SPECIFICATIONS IN AN ARRAY DECLARATION.

MAT - MATRIX OF FORMAL PARAMETERS (REPRESENTED BY THEIR STACK DISPLACEMENT) AND THEIR PROPERTIES.

MATRIX - SAM AS MAT.

MATVEC - AN ENTRY OF MAT OR MATRIX.

MOAMB - USED TO DETECT THE AMBIGUITY ARISING FROM AN EMPTY MODE PART AND/OR AN EMPTY VALUE PART

MODE - THE MODE OF TRANSMISSION OF A FORMAL PARAMETER.

N - NUMBER OF DIMENSIONS OF AN ARRAY, LENGTH OF A SWITCH LIST. A COMPONENT OF A SYMBOL TABLE ENTRY.

NAMETB - TABLE RELATING THE SPELLING OF FORMAL PARAMETERS TO THEIR STACK DISPLACEMENT.

NFORMALS - NUMBER OF FORMAL PARAMETERS. A COMPONENT OF A SYMBOL TABLE ENTRY.

NLOCALS - NUMBER OF ATTRIBUTES OF A CLASS. A COMPONENT OF A SYMBOL TABLE ENTRY.

NOLABEL - DETECTS A MULTILABELLED BLOCK.

NTB - COLLECTS THE ENTRIES FOR NTB.

NUMDEC - USED FOR DISAMBIGUATION PURPOSES. COUNTS THE NUMBER OF EMPTY DECLARATIONS IN A BLOCK HEAD.

O - USED FOR DISAMBIGUATION PURPOSES. USED TO DETECT EMPTY OTHERWISE CLAUSES

OBJECT - SEGMENT DESIGNATION OF THE OBJECT WHICH IS THE CLASS' PROTOTYPE. A COMPONENT OF A SYMBOL TABLE ENTRY.

OPEN - SAME AS O

OPER - CONTAINS OPERANDS. A COMPONENT OF INSTR.

ORIG - SEGMENT THAT CONTAINS THE FIRST INSTRUCTION OF A SIMULA PROGRAM.

OUTERMOST - MARKS A STATEMENT AS THE BODY OF A CLASS.

PL - THE PROPERTY LIST ASSOCIATED WITH AN IDENTIFIER OR EXPRESSION. HAS THE SAME STRUCTURE AS A SYMBOL TABLE ENTRY.

PLACE - GIVES THE CONTEXT OF AN IDENTIFIER LIST.

PPL - PROPERTY LIST OF A PREFIX CLASS IN A CLASS DECLARATION.

PREF - SEGMENT DESIGNATION OF THE PREFIX CLASS. A COMPONENT OF A CLASS' SYMBOL TABLE ENTRY.

PREFIX - A COMPONENT OF AN ENTRY OF QUALTB. POINTS TO THE QUALTB'S ENTRY CORRESPONDING TO THE CLASS' PREFIX.

QTBVEC - AN ENTRY OF QUALTB.

QUAL - THE SEGMENT DESIGNATION OF THE CLASS THAT QQUALIFIES A REFERENCE. USUALLY A COMPONENT OF GENUS.

QUALTB - A TABL GIVING THE PREIX SEQUENCE OF CLASSES, EACH ENTRY CORRESPONDING TO A CLASS AND CHARACTERIZED BY THE SEGMENT DESIGNATION OF THE CLASS.

RULE - A LIST OF INSTRS. THE OBJECT CODE GENERATED FOR THE STRING DERIVED FROM A NONTERMINAL.

SEGMENT - THE SEGMENT DESIGNATION OF A CLASS. USUALLY A COMPONENT OF A SYMBOL TABLE ENTRY.

SID - USED TO DISAMBIGUATE ACTUAL PARAMETERS, ETC.. IDENTIFIES AN EXPRESSION AS A SINGLE ENTITY.

SL, SM, SN - HOLD EITHER A SEGMENT DESIGNATION OR THE UNIQUE INTEGER ASSOCIATED WITH A LABEL.

SP - THE SPELLING OF AN IDENTIFIER.

SPEC - THE GENUS ASSOCIATED WITH AN IDENTIFIER IN AN IDENTIFIER LIST.

START - USED TO DISAMBIGUATE SPLIT BODIES WHOSE FIRST STATEMENT IS EMPTY.

T - THE TYPE OF THE PRODUCT IN A MULTIPLICATION.

TYPD - THE GENUS OF AN IDENTIFIER BEING DECLARED.

TYPDS - THE GENUS OF A SPECIFIER.

TYPE - THE TYPE OF AN IDENTIFIER.

TJUMP - SIMILAR TO FJUMP.

UNDECL - SAME STRUCTURE AS RULE. COLLECTS THE INSTRUCTIONS GENERATED BY THE DECLARATION OF LABELS.

USE - USE OF AN EXPRESSION: FOR ITS VALUE, ITS LOCATION OR FOR LATER EXECUTION.

V - THE VALUE OF AN INTEGER.

VALENCE - CLASSIFIES "+" OR "-" AS EITHER A UNARY OR BINARY OPERATOR.

VIRDECL - COLLECTS THE INSTRS THAT REPLACE THE VIRTUAL ATTRIBUTES THAT HAVE BEEN REDECLARED.

VIRTUALE - SYMBOL TABLE FORMED BY THE VIRTUAL ATTRIBUTES OF A CLASS. A COMPONENT OF A SYMBOL TABLE ENTRY.

XX - A COMPONENT OF ITEM.

IDENTIFIERS ARE SIGMA WITH ATTRIBUTE SP

INTEGERS ARE NU WITH ATTRIBUTE V

COMMENT THE FOLLOWING IS A LIST OF THE ABBREVIATIONS USED FOR THE NONTERMINAL IDENTIFIERS AND THE PRODUCTION WHICH FIRST FINDS THEM ON THE LEFT HAND SIDE:

ABBREVIATION	NONTERMINAL	PRODUCTION
AP	ACTUAL PARAMETER	P95
APLIST	ACTUAL PARAMETER LIST	P93
APPART	ACTUAL PARAMETER PART	P91
AOP	ADDING OPERATOR	P12
ARITEXPR	ARITHMETIC EXPRESSION	P7
ARDECL	ARRAY DECLARATION	P206
ARID	ARRAY IDENTIFIER	P212
ARID1	ARRAY IDENTIFIER'	P85
ARLIST	ARRAY LIST	P208
ARSEG	ARRAY SEGMENT	P210
ASSST	ASSIGNMENT STATEMENT	P283
ATTRID	ATTRIBUTE IDENTIFIER	P84
BASICST	BASIC STATEMENT	P273
BLOCK	BLOCK	P174
BLOCKHEAD	BLOCK HEAD	P184
BLOCKPRE	BLOCK PREFIX	P181
BEXPR	BOOLEAN EXPRESSION	P103
BFAC	BOOLEAN FACTOR	P111
BPRIM	BOOLEAN PRIMARY	P115
BSEC	BOOLEAN SECONDARY	P113
BTERM	BOOLEAN TERM	P109
BOUND	BOUND	P216
BOUNDP	BOUND PAIR	P215
BOUNDPLIST	BOUND PAIR LIST	P213

CLBODY	CLASS BODY	P256
CLDECL	CLASS DECLARATION	P250
CLID	CLASS IDENTIFIER	P255A
CLID1	CLASS IDENTIFIER'	P158
CLID2	CLASS IDENTIFIER''	P157A
COMPST	COMPOUND STATEMENT	P171
COMPT	COMPOUND TAIL	P264
CONDST	CONDITIONAL STATEMENT	P298
CONNBLOCK1	CONNECTION BLOCK'	P313
CONNBLOCK2	CONNECTION BLOCK''	P314
CONNCL	CONNECTION CLAUSE	P312
CONNPART	CONNECTION PART	P310
CONNST	CONNECTION STATEMENT	P307
DECL	DECLARATION	P186
DESIGEXPR	DESIGNATIONAL EXPRESSION	P161
DUMMYST	DUMMY STATEMENT	P296
EXPR	EXPRESSION	P1
FAC	FACTOR	P19
FINOPS	FINAL OPERATIONS	P262
FP	FORMAL PARAMETER	P229
FPLIST	FORMAL PARAMETER LIST	P227
FPPART	FORMAL PARAMETER PART	P225
FUNC	FUNCTION DESIGNATOR	P89
GOTOST	GO TO STATEMENT	P295
ID1	IDENTIFIER'	P52
IDLIST	IDENTIFIER LIST	P234
IFCL	IF CLAUSE	P304
IFST	IF STATEMENT	P303
IMPL	IMPLICATION	P107
INITOPS	INITIAL OPERATIONS	P259
LABEL0	LABEL	P172A
LABEL1	LABEL'	P166
LOCOBJ	LOCAL OBJECT	P159
LOGVAL	LOGICAL VALUE	P120
MBLOCK	MAIN BLOCK	P182
MPART	MAIN PART	P253
MOPART	MODE PART	P230
MOP	MULTIPLICATION OPERATOR	P16
NAMEPART	NAME PART	P236
OBJEXPR	OBJECT EXPRESSION	P148
OBJGEN	OBJECT GENERATOR	P157
OBJREF	OBJECT REFERENCE	P200
OBJREFREL	OBJECT REFERENCE RELATION	P136
OBJREL	OBJECT RELATION	P133
OTCL	OTHERWISE CLAUSE	P315
PRE	PREFIX	P251
PRIM	PRIMARY	P21
PROCBODY	PROCEDURE BODY	P247
PRODECL	PROCEDURE DECLARATION	P222
PROCHEAD	PROCEDURE HEADING	P223
PROCID	PROCEDURE IDENTIFIER	P224
PROCID1	PROCEDURE IDENTIFIER'	P90
PROCID2	PROCEDURE IDENTIFIER''	P287A
PROCST	PROCEDURE STATEMENT	P297
PROGRAM	PROGRAM	P169
QUALIF	QUALIFICATION	P201
QUALOBJ	QUALIFIED OBJECT	P160
REL	RELATION	P122

Reproduced from best available copy.

RELOP	RELATIONAL OPERATOR	P126
REFASS	REFERENCE ASSIGNMENT	P290
REFCOMP	REFERENCE COMPARATOR	P137
REFEXPR	REFERENCE EXPRESSION	P147
REFLPART	REFERENCE LEFT PART	P291
REFRPART	REFERENCE RIGHT PART	P293
REFREL	REFERENCE RELATION	P135
REFTYPE	REFERENCE TYPE	P199
RID	REMOTE IDENTIFIER	P83
SARITEXPR	SIMPLE ARITHMETIC EXPRESSION	P9
SBOOL	SIMPLE BOOLEAN	P105
SDESIGEXPR	SIMPLE DESIGNATIONAL EXPRESSION	P163
SOBJEXPR	SIMPLE OBJECT EXPRESSION	P150
SPPART	SPECIFICATION PART	P238
SPECIFIER	SPECIFIER	P240
SPLITBODY	SPLIT BODY	P258
ST	STATEMENT	P265A
ST1	STATEMENT'	P266
SUBEXPR	SUBSCRIPT EXPRESSION	P88
SUBLIST	SUBSCRIPT LIST	P86
SWDECL	SWITCH DECLARATION	P218
SWDESIG	SWITCH DESIGNATOR	P167
SWID	SWITCH IDENTIFIER	P218A
SWID1	SWITCH IDENTIFIER'	P168
SWLIST	SWITCH LIST	P219
TERM	TERM	P14
TYPEN	TYPE	P193
TYPEP	PROCEDURE TYPE	P221A
TYPEDECL	TYPE DECLARATION	P192
TYPELIST	TYPE LIST	P202
UNCONDST	UNCONDITIONAL STATEMENT	P270
UNLBASICST	UNLABELLED BASIC STATEMENT	P275
UNLBLOCK	UNLABELLED BLOCK	P179
UNLCOMP	UNLABELLED COMPOUND	P173
UNLPREBLOCK	UNLABELLED PREFIXED BLOCK	P180
VALASS	VALUE ASSIGNMENT	P385
VALEXPR	VALUE EXPRESSION	P4
VALLPART	VALUE LEFT PART	P286
VALPART	VALUE PART	P232
VALRPART	VALUE RIGHT PART	P288
VALTYPE	VALUE TYPE	P195
VAR	VARIABLE	P48
VIRPART	VIRTUAL PART	P254
WHILEST	WHILE STATEMENT	P305;

NONTERMINALS ARE

AP = S(RULE), I(ENV, ITEM, QUALTB, LL)
APLIST = S(L, RULE), I(ENV, ITEM, QUALTB, LL)
APPART = S(L, RULE), I(ENV, ITEM, QUALTB, LL)

```
AOP = S(RULE), I(VALENCE)
ARITEXPR = S(PL, RULE, SID), I(ENV, QUALTB, ITEM, LL, USE, APA)
ARDECL = S(D, E, RULE), I(ENVA, QUALTB, CL, DO, ITEM, LL, ENV)
ARID = S(SP)
ARID1 = S(PL, RULE, SP), I(ENV, USE, ITEM, LL, APA, QUALTB)
ARLIST = S(D, E, RULE), I(DO, ENVA, ITEM, LL, TYPD, QUALTB, CL)
ARSEG = S(D, E, L, RULE),
        I(CL, DO, ENVA, ITEM, LL, TYPD, QUALTB)
ASSST = S(RULE), I(ENV, ITEM, LL, QUALTB)
ATTRID = S(PL, SP), I(ENV)
BASICST = S(D, E, RULE, UNDECL, VIRDECL, FIRSTST),
          I(LL, DO, ENV, ITEM, ENV1, QUALTB, CL)
BLOCK = S(D, E, RULE, UNDECL, VIRDECL, NOLABEL),
        I(CL, DO, ENV, ENV1, ENVA, ITEM, LL, BEGUN, QUALTB)
BLOCKHEAD = S(D, E, RULE, NUMDEC, AEMDEC, EMDEC, VIRDECL, CDECL),
            I(CL, DO, ENV, ENV1, ENVA, ITEM, LL, QUALTB)
BLOCKPRE = S(PL, RULE), I(ENV, ITEM, LL, QUALTB)
BEXPR = S(PL, RULE, SID), I(ENV, ITEM, LL, QUALTB, USE, APA)
BFAC = S(PL, RULE, SID), I(ENV, ITEM, LL, QUALTB, USE, APA)
BOUND = S(RULE), I(ENVA, ITEM, QUALTB, LL)
BPRIM = S(PL, RULE, SID), I(ENV, ITEM, LL, QUALTB, USE, APA)
BSEC = S(PL, RULE, SID), I(ENV, ITEM, LL, QUALTB, USE, APA)
BTERM = S(PL, RULE), I(ENV, ITEM, LL, QUALTB, USE, APA)
BOUNDP = S(RULE), I(ENVA, ITEM, LL, QUALTB)
BOUNDPLIST = S(L,RULE), I(ENVA, ITEM, LL, QUALTB)
CLBODY = S(D, E, RULE, UNDECL, VIRDECL),
         I(CL, DO, ENV, ENV1, ENVA, ITEM, LL, QUALTB)
CLDECL = S(D, E, RULE), I(CL, DO, ENV, ITEM, QUALTB, LL)
CLID = S(SP)
CLID1 = S(PL, SP), I(ENV)
CLID2 = S(PL, RULE, SP), I(ENV, ITEM, QUALTB, LL, USE)
COMPST = S(E, RULE, UNDECL, VIRDECL, D),
         I(DO, ITEM, ENV1, ENV, LL, QUALTB, CL)
COMPT = S(E, RULE, UNDECL, VIRDECL, D, FIRSTST),
        I(QUALTB, ENV, ENV1, DO, LL, ITEM, CL)
CONDST = S(D, E, OPEN, RULE, UNDECL, VIRDECL),
         I(LL, DO, ENV, ENV1, ITEM, QUALTB, CL)
CONNBLOCK1 = S(D, E, OPEN, RULE, UNDECL),
             I(DO, ENV, ITEM, LL, BEGUN, QUALTB)
CONNBLOCK2 = S(D, E, OPEN, RULE, UNDECL),
             I(DO, ENV, ITEM, LL, BEGUN, QUALTB)
CONNCL = S(OPEN, RULE), I(ENV, ITEM, LL, FJUMP, TJUMP, QUALTB)
CONNPART = S(OPEN, RULE), I(ENV, ITEM, LL, FJUMP, TJUMP, QUALTB)
CONNST = S(D, E, OPEN, RULE, UNDECL, VIRDECL),
         I(DO, ENV, ENV1, ITEM, LL, QUALTB, CL)
DECL = S(D, E, RULE, VIRDECL, CDECL, EMDEC),
       I(CL, DO, ENV, ITEM, LL, ENV1, QUALTB, ENVA)
DESIGEXPR = S(RULE), I(ENV, ITEM, QUALTB, APA, LL)
DUMMYST = S(RULE)
EXPR = S(PL, RULE), I(ENV, ITEM, LL, QUALTB, APA, USE)
FAC = S(PL, RULE, SID), I(ENV, ITEM, LL, QUALTB, APA, USE)
FINOPS = S(E, RULE, UNDECL, VIRDECL, D),
         I(LL, DO, ENV, ENV1, ITEM, QUALTB, CL)
FP = S(SP)
FPLIST = S(D, NTB), I(DO)
FPPART = S(D, NTB), I(DO)
FUNC = S(L, PL, RULE, SP), I(ENV, ITEM, QUALTB, LL, APA)
GOTOST = S(RULE), I(ENV, ITEM, QUALTB, LL)
```


Reproduced from best available copy.

```
ID = S(SP)
ID1 = S(PL, RULE, SP), I(ENV, ITEM, QUALTB, LL, USE)
IDLIST = S(MATRIX, L, E),
         I(CL, ENV1, MAT, NAMETB, LL, DO, TYPD, PLACE)
IFCL = S(RULE), I(ENV, ITEM, QUALTB, LL, FJUMP)
IFST = S(D, E, RULE, UNDECL, VIRDECL),
       I(ENV, ITEM, QUALTB, LL, DO, ENV1, FJUMP)
IMPL = S(PL, RULE, SID), I(ENV, ITEM, QUALTB, LL, USE, APA)
INITOPS = S(D, E, RULE, UNDECL, VIRDECL, START, ARULE, EMDEC),
          I(CL, DO, ENV, ENV1, ENVA, ITEM, QUALTB, LL)
LABEL0 = S(SP)
LABEL1 = S(SP, PL, RULE), I(ENV, APA)
LOCOBJ = S(PL, RULE), I(ENV)
LOGVAL = S(RULE)
MBLOCK = S(D, E, RULE, UNDECL, VIRDECL),
         I(CL, ENV, ENVA, ENV1, ITEM, QUALTB, LL, DO, BEGUN)
MPART = S(E, PL, RULE, VIRDECL, SP),
        I(CL, ENV, ITEM, QUALTB, PPL, LL)
MOPART = S(MATRIX), I(MAT, NAMETB)
MOP = S(PL, RULE), I(E)
NAMEPART = S(MATRIX, NOAMB), I(MAT, NAMETB)
OBJEXPR = S(PL, RULE, SID), I(ENV, ITEM, QUALTB, LL, USE, APA)
OBJGEN = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE)
OBJREF = S(TYPDS), I(ENV)
OBJREFREL = S(RULE), I(ENV, ITEM, QUALTB, LL, USE)
OBJREL = S(RULE), I(ENV, ITEM, QUALTB, LL, USE)
OTCL = S(OPEN, RULE, UNDECL, VIRDECL, D, E),
       I(LL, DO, ENV, ENV1, ITEM, QUALTB, CL, O)
PRE = S(PL), I(ENV, LL)
PRIM = S(PL, RULE, SID), I(ENV, ITEM, QUALTB, LL, USE, APA)
PROCBODY = S(RULE, UNDECL, E),
           I(DO, ENV, ENVA, ENV1, ITEM, LL, QUALTB)
PROCDECL = S(D, E, RULE, VIRDECL),
           I(CL, DO, ENV, ENV1, ITEM, QUALTB, LL)
PROCHEAD = S(D, E, RULE, SP), I(TYPD, LL, ENV)
PROCID = S(SP)
PROCID1 = S(PL, SP, RULE), I(ENV, ITEM, QUALTB, LL, USE, APA)
PROCID2 = S(PL, SP), I(ENV)
PROCST = S(RULE), I(ENV, ITEM, QUALTB, LL)
PROGRAM = S(ORIG)
QUALIF = S(PL), I(ENV)
QUALOBJ = S(PL, RULE), I(ENV, QUALTB, ITEM, LL, USE)
REL = S(RULE, I(ENV, ITEM, QUALTB, LL, USE)
RELOP = S(RULE)
REFASS = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, ALSO)
REFCOMP = S(RULE)
REFEXPR = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE, APA)
REFLPART = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE)
REFRPART = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE, ALSO)
REFREL = S(RULE), I(ENV, ITEM, QUALTB, LL, USE)
REFTYPE = S(TYPDS), I(ENV)
RID = S(PL, RULE, SP), I(ENV, ITEM, QUALTB, LL, USE)
SARITEXPR = S(PL, RULE, SID), I(ENV, ITEM, QUALTB, LL, USE, APA)
SBOOL = S(PL, RULE, SID), I(ENV, ITEM, QUALTB, LL, USE, APA)
SDESIGEXPR = S(RULE), I(ENV, ITEM, QUALTB, LL, APA)
SOBJEXPR = S(PL, RULE, SID), I(ENV, ITEM, QUALTB, LL, APA, USE)
SPPART = S(E, L, RULE, MATRIX),
         I(NAMETB, CL, DO, ENV1, LL, LEGIT, PLACE)
```

```
SPECIFIER = S(TYPDS), I(ENV)
SPLITBODY = S(D, E, RULE, UNDECL, VIRDECL),
            I(CL, DO, ENV, ENV1, ENVA, ITEM, QUALTB, LL)
ST = S(D, E, OPEN, RULE, UNDECL, VIRDECL, FIRSTST),
     I(DO, ENV, ENV1, ITEM, LL, QUALTB, CL)
ST1 = S(D, E, OPEN, RULE, UNDECL, VIRDECL, FIRSTST),
      I(CL, DO, ENV, ENV1, ENVA, ITEM, OUTERMOST, QUALTB, LL, BEGUN)
SUBEXPR = S(RULE), I(ENV, ITEM, LL, QUALTB, USE)
SUBLIST = S(L, RULE), I(ENV, ITEM, LL, QUALTB)
SWDECL = S(D, E, RULE, VIRDECL),
         I(CL, DO, ENV, ENV1, ITEM, LL, QUALTB)
SWDESIG = S(RULE), I(ENV, ITEM, QUALTB, LL, APA)
SWID = S(SP)
SWID1 = S(PL, RULE, SP), I(ENV, APA)
SWLIST = S(L, RULE), I(ENV, ITEM, LL, QUALTB)
TERM = S(PL, RULE, SID), I(ENV, ITEM, LL, QUALTB, USE, APA)
TYPEN = S(TYPDS), I(ENV)
TYPEP = S(TYPDS), I(ENV)
TYPEDECL = S(D, E, RULE), I(CL, DO, ENV, LL)
TYPELIST = S(D, E), I(CL, DO, TYPD, LL)
UNCONDST = S(D, E, RULE, UNDECL, VIRDECL, FIRSTST),
           I(CL, DO, ENV, ENV1, ENVA, ITEM, OUTERMOST, QUALTB, LL,
             BEGUN)
UNLBASICST = S(RULE, FIRSTST), I(ENV, ITEM, LL, QUALTB, CL)
UNLBLOCK = S(D, E, RULE, UNDECL, VIRDECL),
           I(CL, DO, ENV, ENV1, ENVA, ITEM, QUALTB, LL, BEGUN)
UNLCOMP = S(D, E, RULE, UNDECL, VIRDECL),
          I(DO, ENV, ENV1, ITEM, QUALTB, LL, CL)
UNLPREBLOCK = S(D, E, RULE, UNDECL, VIRDECL),
              I(ENV, ITEM, LL, QUALTB)
VALASS = (PL, RULE), I(ENV, ITEM, QUALTB, LL, ALSO)
VALEXPR = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE, APA)
VALLPART = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE)
VALPART = S(MATRIX, NOAMB), I(MAT, NAMETB)
VALRPART = S(PL, RULE), I(ENV, ITEM, QUALTB, LL, USE, ALSO)
VALTYPE = S(TYPDS)
VAR = S(PL, RULE, SP, DAR), I(ENV, ITEM, QUALTB, LL, USE, APA)
VIRTPART = S(E, L, RULE), I(CL, DO, ENV1, LL)
WHILEST = S(D, E, OPEN, RULE, UNDECL, VIRDECL),
          I(LL, DO, ENV, ENV1, ITEM, QUALTB, CL)


START SYMBOL PROGRAM


FORMATS ARE

F1 = ("GO(", DISP, ")")
F2 = ("AR(", OPER, ")")
F3 = ("C(INTEGER(VALUE=", V, "))")
F4 = ("INX(", USE,")")
F5 = ("VAL(ADDR=(", ADDR, "), REM)")
F6 = ("ADR(ADDR=(", ADDR, "), REM)")
F7 = ("VAL(ADDR=(", ADDR, "))")
F8 = ("ADR(ADDR=(", ADDR, "))")
F9 = ("C(", OPER, ")")
```

```
F10 = ("ENT")
F11 = ("MARK")
F12 = ("RET")
F13 = ("C(ACTUAL(BODY=", SN, ",LEVEL=", LL, ",QUAL=", QUAL,
                    ",UNDERTYPE=", TYPE, "))")
F14 = ("LOG(", OPER, ")")
F15 = ("C(BOOLEAN(VALUE=", OPER, "))")
F16 = ("COMP(", OPER, ")")
F17 = ("IFJ(", DISP, ")")
F18 = ("C(REF(QUAL=", QUAL, ",VALUE=",OPER,"))")
F19 = ("GEN(",NFORMALS,")")
F20 = ("C(LABEL(SEGMENT=",SN,",DISP=",DISP,"))")
F21 = ("LN=",LN,",DN=",DN)
F22 = ("ENT(LEVEL=",LL,",BODY=",SN,")")
F23 = ("MAK(GENUS=(",GENUS,"),N=",L,"COPIES=",D,")")
F24 = ("C(SWITCHLIST=",SN,",LENGTH=",L,")")
F25 = ("DEL")
F26 = ("RES")
F27 = ("DET")
F28 = ("STO(",ALSO,")")
F29 = ("GO")
F30 = ("C(RET)")
F31 = ("NEW OBJECT(BODY=",SN,"IS=",SM,",PREFIX=",OBJECT,")")
F32 = ("C(CLASS(PROTOTYPE=",SN,",LEVEL=",LL,"))")
F33 = ("C(PROCEDURE(LEVEL=",LL",SEGMENT=",SN,"))")
F34 = ("CHE(D=",D,",GENUS=(",GENUS,"),MODE=",MODE,",CLASS=",ALSO,")")
F35 = ("DET(TER)")
F36 = ("KIND=",KIND,",TYPE=",TYPE,",QUAL=",QUAL)
```

```
PROCEDURE AUX (CLASSN, QUALTB);
COMMENT   THIS PROCEDURE DOES THE WORK FOR UPDQUALTB BY ACTUALLY
   INSERTING THE NEW ENTRIES;

$/ QUAL := [CLASSN]. PREF;
   %PUTIN(QUALTB.[[CLASSN].SEGMENT]: CLASSN := CLASSN;
          PREFIX := IF QUAL = 0 THEN NIL ELSE QUALTB.[QUAL];
          LEVEL := IF QUAL = 0 THEN 0 ELSE QUALTB.[QUAL].LEVEL+1) /$;
```

```
FUNCTION BACTUAL (GENUS, LL, SN);
BEGIN COMMENT THIS FUNCTION WILL BUILD AND RETURN A RULE WITH AN
        INSTRUCTION C(ACTUAL) WITH THE PROPER OPERANDS;

   INSTR := $(FORMAT := F13; SN := SN; LL := LL + 1;
                                  TYPE := GENUS.TYPE);
   LIST(IF NULLR(FIND(GENUS, QUAL)) THEN INSTR ELSE
                              PUTIN(INSTR : QUAL := GENUS.QUAL))
END;
```

```
FUNCTION BUILDVC (KIND, L);
BEGIN COMMENT THIS PRODUCES A LIST OF L C-INSTRUCTIONS, ALL OF KIND
         "KIND", FOR P239;

   RULE := NULL;
   WHILE L>0 DO
   BEGIN
      RULE := CONS(INSTR ** $(FORMAT := F9;  OPER := KIND), RULE);
      L := L - 1
   END;
   RULE
END;



FUNCTION CHECKIDENTIFIER (ITEM, QUALTB, ATTR);
BEGIN COMMENT THIS CHECKS TO SEE IF ANY OF THE COMPONENTS XX OF
         ITEM CONTAIN A QUALIFICATION SUCH THAT XX.QUAL IN ATTR.  IF
         TRUE THE ADDR OF THE CORRESPONDING XX IS RETURNED OTHERWISE
         NULL IS RETURNED. THIS FUNCTION IS USED TO LOCATE VARIABLES
         THAT ARE IN THE STACK OF ANOTHER OBJECT TO WHICH THE PRESENT
         OBJECT IS CONNECTED. THE LIST ITEM CONTAINS THE ADDR OF
         WORDS IN THE STACK THAT REFERENCES OBJECTS CONNECTED TO THIS
         ONE;

   IF ATTR = 0 OR NULLB(ITEM) THEN NULL ELSE
   BEGIN
      NEXT1 := QUALTB.[ATTR];  LEVEL := [NEXT1].LEVEL;
      XX := CAR(ITEM);
      WHILE ¬NULLB(XX) DO
      BEGIN
         NEXT2 := QUALTB.[XX.QUAL];  N1 := [NEXT2].LEVEL - LEVEL;
         IF N1 >= 0 THEN
         BEGIN
            WHILE N1 > 0 DO
            BEGIN
               NEXT2 := [[NEXT2].PREFIX];  N := N-1
            END;
            IF NEXT1 = NEXT2 THEN GO TO FINISH
         END;
         ITEM := CDR(ITEM); XX := CAR(ITEM)
      END;
FINISH : ;  XX.ADDR
   END
END;



FUNCTION CHECKKIND (GENUS);
BEGIN COMMENT THIS FUNCTION IS USED TO CHECK IF FORMAL PARAMETERS
         HAVE THE PROPER MODE. THE RESULT IS A BOOLEAN;

   KIND := GENUS.KIND;
   KIND = "LABEL" OR KIND = "SWITCH" OR KIND = "PROCEDURE"
END;
```

```
PROCEDURE CHECKSPEC (MATRIX, D);
BEGIN COMMENT THIS PROCEDURE WILL CHECK TO SEE IF ALL D FORMAL
        PARAMETERS OF MATRIX HAVE BEEN SPECIFIED;

   NEXT1 := FIRST(MATRIX);
   WHILE ¬NULLB(NEXT1) DO
   BEGIN NEXT1 := NEXT([NEXT1]);   D := D - 1 END;
   IF D >= 0 THEN
   ERROR("PROCEDURE OR CLASS HAS UNSPECIFIED FORMAL PARAMETERS")
END;


FUNCTION CHECKVIRT (ENV1, SP, OPER);
BEGIN COMMENT THIS CHECKS IF SP IS AN ENTRY IN VIRTUALE:  IF TRUE
        IT RETURNS ADDR.DN, IF NOT ZERO;

   NEXT1 := FIND(ENV1, [SP]);
   IF NULLB(NEXT1) THEN DN := 0 ELSE
   IF [NEXT1].GENUS.KIND = OPER THEN DN := [NEXT1].ADDR.DN ELSE
   ERROR(SP, " HAS BEEN DECLARED TWICE, ONCE AS A VIRTUAL");
   DN;
END;


FUNCTION CHERULES (MATRIX, ALSO, D, DO);
BEGIN COMMENT THIS WILL BUILD THE SEQUENCE OF CHE INSTRUCTIONS
        THAT HEAD THE RULE FOR A PROCEDURE OR A CLASS;

   RULE := NULL; NEXT1 := FIND(MATRIX, [D]);
   WHILE ¬NULLB(NEXT1) DO
   BEGIN
       RULE := CONS(INSTR *= $(FORMAT := F34;  ALSO := ALSO;
                              GENUS := [NEXT1].SPEC := $(FORMAT := F36);
                              D := D -DO;  MODE := [NEXT1].MODE), RULE);
       D := D-1; NEXT1 := FIND(MATRIX, [D])
   END;
   RULE
END;


FUNCTION COMBTYPE (PL, PL1);
BEGIN COMMENT THIS WILL EXAMINE THE TYPE OF BOTH PL↑S AND IF BOTH
         ARE NOT "INTEGER" IT RETURNS "REAL" OTHERWISE THE VALUE OF
         THE COMPONENT TYPE OF PL1 IS RETURNED.  THE VALUES ARE
         EITHER "INTEGER" OR "REAL";

   IF PL.GENUS.TYPE ¬= "INTEGER" THEN "REAL" ELSE PL1.GENUS.TYPE

END;


FUNCTION CONCATENATE (RULE1, RULE2);
BEGIN COMMENT THIS WILL CONCATENATE TWO RULES, ONE REPRESENTING THE
         PREFIX PART AND THE OTHER THE MAIN PART OF A CLASS BODY.
```

```
        THE CONCATENATION IS DONE IN THE FOLLOWING FORM (USING
        WILNER'S NOTATION): CHE(P) C(P) CHE(M) C(M) INIT I(P) I(M)
        INNER F(M) F(P). WHILE THE LIST RULE2 IS USED DIRECTLY,
        RULE1 IS COPIED BY BUILDING THE NECESSARY NEW LISTS;

  COMMENT FIRST BUILD A COPY OF CHE(P) AND C(P) PARTS OF RULE.
     THE FIRST PSEUDO-INSTRUCTION MARK FOUND IS THE INIT OF RULE1;

  RULE := NULL;
  WHILE NULLR(FIND(CAR(RULE1), MARK)) DO
  BEGIN
     RULE := CONS(CAR(RULE1), RULE); RULE := CDR(RULE1)
  END;
  COMMENT NOW WE COPY CHE(M) AND C(M);

  WHILE NULLR(FIND(CAR(RULE2), MARK)) DO
  BEGIN
     RULE := CONS(CAR(RULE2), RULE);  RULE2 := CDR(RULE2)
  END;
  COMMENT INSERT INIT IN THE NEW RULE;

  RULE := CONS(CAR(RULE1) RULE);  RULE1 := CDR(RULE1);
  COMMENT COPY THE I(P). THE END OF I(P) IS MARKED BY A MARK
      PSEUDO-INSTRUCTION;

  WHILE ¬NULLR(FIND(CAR(RULE1),MARK)) DO
  BEGIN
    RULE := CONS(CAR(RULE1), RULE); RULE1 := CDR(RULE1);
  END;
  COMMENT NOW WE REVERSE RULE AND APPEND I(M) INNER F(M) WHICH IS
  NOW CDR(RULE2) (WE HAVE TO ELIMINATE THE INIT) AND THEN REVERSE
  IT. AGAIN TAKE THE CDR ( TO ELIMINATE THE DET INSTRUCTION AT THE
  END OF RULE2) AND THEN COPY F(P). THE REASON WE HAVE TO COPY F(P)
  AND CANNOT SIMPLY APPEND IT IS THAT THE JUMP INSTRUCTIONS
  (GO & IFJ) WILL HAVE DIFFERENT DISPLACEMENT VALUES DEPENDING ON
  THE CODE SEGMENT;

  RULE := CDR(RVRS(APEND(RVRS(RULE), CDR(RULE2))));
  RULE1 := CDR(RULE1);
  WHILE ¬NULLB(RULE1) DO
  BEGIN
     RULE := CONS(CAR(RULE1). RULE);  RULE1 := CDR(RULE1)
  END;
  RVRS(RULE)
END;


FUNCTION CONDQUAL (QUALTB, QUAL1, QUAL2);
BEGIN  COMMENT THIS TAKES THE QUALIFICATIONS OF TWO CLASSES
           AND OUTPUTS THE QUALIFICATION OF THE CLASS WHICH IS
           THE LAST IN THEIR PREFIX SEQUENCE THAT IS COMMON TO
           BOTH;

   COMMENT IF ONE OF THEM IS "NONE" THE RESULT IS THE QUALIFICATION
      OF THE OTHER;

   IF QUAL1 = 0 THEN QUAL2 ELSE
```

Reproduced from best available copy.

```
   IF QUAL2 < 0 THEN QUAL1 ELSE
   BEGIN
      NEXT1 := QUALTB.[QUAL1]; NEXT2 := QUALTB.[QUAL2];
      N1 := [NEXT1].LEVEL; N2 := [NEXT2].LEVEL;
      COMMENT NOW IF N1 > N2 WE INVERT THE TWO AND ALSO NEXT;

      IF N1 > N2 THEN
      BEGIN
         N := [N1];  N1 := [N2]; N2 := [N];
         NEXT3 := [NEXT1];  NEXT1 := [NEXT2];  NEXT2 := [NEXT3]
      END;
      COMMENT NOW IF N1 ≠ N2 WE TAKE THE ANCESTOR OF NEXT2 UNTIL
         N1 = N2;

      WHILE N1 ¬= N2 DO
      BEGIN
         NEXT2 := [[NEXT2].PREFIX];  N2 := N2 -1
      END;
      COMMENT NOW WE LOOK FOR THE COMMON ANCESTOR

      WHILE NEXT1 ¬= NEXT2 DO
      BEGIN
         NEXT1 := [[NEXT1].PREFIX];  NEXT2 := [[NEXT2].PREFIX];
         IF NULLB(NEXT1) THEN ERROR( "NO COMMON ANCESTOR")
      END;
      SELECTOR([NEXT1])
   END
END;


PROCEDURE DISAMV (SP, DAR, COND, PL, APA);
BEGIN COMMENT THIS CHECKS FOR AMBIGUITIES AND DISAMBIGUATES NODES.
         IT IS CALLED BY P22 AND P116. THE AMBIGUITIES RESULT FROM
         ACTUAL PARAMETERS THAT PARSE TO A SINGLE ENTITY. AMBIGUITIES
         ALSO ARISE WHEN THE RHS OF A VALUE ASSIGNMENT PARSES TO A
         A SINGLE ENTITY AND WHEN A PRIMARY PARSES TO AN IDENTIFIER;

   TYPE := PL.GENUS.TYPE;  KIND := PL.GENUS.KIND;
   IF APA = 4 THEN DAMB(KIND ¬= "PROCEDURE", PRIM) ELSE
   IF ¬ DAR THEN
   BEGIN COMMENT THIS IS NOT AN ARRAY;

      IF APA = 0 THEN
      BEGIN COMMENT THIS IS AMBIGUOUS BECAUSE EVERY PRIMARY CAN BE A
               VARIABLE OR A FUNCTION DESIGNATOR WITH NO PARAMETERS;

         IF KIND = "SIMPLE" THEN
         BEGIN
            IF ¬ COND THEN DAMB(TRUE, PRIM) ELSE
            ERROR(SP, " IS OF THE WRONG TYPE")
         END ELSE
         IF KIND = "PROCEDURE" THEN DAMB(FALSE, PRIM) ELSE
         ERROR(SP, " IS OF THE WRONG KIND")
      END ELSE
      IF APA = 1 THEN
      BEGIN COMMENT THIS IS THE AMBIGUITY DUE TO THE RIGHT HAND SIDE
               OF A VALUE ASSIGNMENT;
```

```
      IF KIND = "SIMPLE" THEN
      BEGIN
         IF COND THEN
         BEGIN
            IF TYPE ¬= "REF" THEN DAMB(FALSE,VALEXPR) ELSE
            ERROR(SP," IS OF THE WRONG TYPE")
         END ELSE
         BEGIN DAMB(TRUE, PRIM); DAMB(TRUE, VALEXPR) END
      END ELSE
      IF KIND = "PROCEDURE" THEN DAMB(FALSE, PRIM) ELSE
      ERROR(SP, " IS OF THE WRONG-KIND")
   END ELSE
   IF APA = 3 THEN
   BEGIN COMMENT THIS IS AN ACTUAL PARAMETER AMBIGUITY;

      IF KIND ¬= "SIMPLE" THEN
      BEGIN
         IF KIND = "LABEL" THEN DAMB(FALSE,EXPR)
                           ELSE DAMB(FALSE,AP)
      END ELSE
      IF COND THEN
      BEGIN
         IF TYPE ¬= "REF" THEN DAMB(FALSE, VALEXPR)
                          ELSE DAMB(FALSE, EXPR)
      END ELSE
      BEGIN
         DAMB(TRUE, PRIM);  DAMB(TRUE,VALEXPR); DAMB(TRUE, EXPR);
         DAMB(TRUE,AP)
      END
   END ELSE
   COMMENT  THIS IS AN ACTUAL PARAMETER WHICH IS ENCLOSED IN
      PARENTHESES;

   IF KIND = "LABEL" THEN DAMB(FALSE,EXPR) ELSE
   IF KIND = "SWITCH" OR KIND = "ARRAY" THEN
   ERROR(SP, " IS OF THE WRONG KIND") ELSE
   IF COND THEN
   BEGIN
      IF TYPE = "REF" THEN DAMB(FALSE, EXPR)
                      ELSE DAMB(FALSE, VALEXPR)
   END ELSE
   IF KIND = "PROCEDURE" THEN DAMB(FALSE, PRIM) ELSE
   BEGIN DAMB(TRUE, PRIM); DAMB(TRUE, VALEXPR); DAMB(TRUE, EXPR)
   END
END ELSE
COMMENT IT IS AN ARRAY;

IF APA = 0 THEN
BEGIN COMMENT NO AMBIGUITY HERE;

   IF COND OR KIND ¬= "ARRAY THEN
   ERROR(SP, " HAS WRONG TYPE OR KIND")
END ELSE
IF APA = 1 THEN
BEGIN COMMENT THIS IS THE RIGHT HAND SIDE AMBIGUITY;

   IF KIND = "ARRAY" THEN DAMB(¬COND, VALEXPR) ELSE
```

```
      ERROR(SP, " IS OF THE WRONG KIND")
   END ELSE
   COMMENT THIS IS THE ACTUAL PARAMETER AMBIGUITY;

   IF KIND ¬= "ARRAY" THEN
   BEGIN
      IF KIND = "SWITCH" THEN DAMB(FALSE, EXPR) ELSE
      ERROR(SP, " IS OF THE WRONG KIND")
   END ELSE
   IF COND THEN
   BEGIN
      IF TYPE ¬= "REF" THEN DAMB(FALSE, VALEXPR)
                       ELSE DAMB(FALSE, EXPR)
   END ELSE
   BEGIN DAMB(TRUE, VALEXPR);  DAMB(TRUE, EXPR) END
END;


PROCEDURE DISAMF (SP, COND, APA, L);
BEGIN COMMENT THIS IS SIMILAR TO DISAMV BUT HERE THE ARRAY PROBLEM
        DOES NOT ARISE;

   KIND := PL.GENUS.KIND;  TYPE := PL.GENUS.TYPE;
   IF APA = 4 THEN DAMB(KIND ¬= "SIMPLE", PRIM)
   IF L = 0 THEN
   BEGIN COMMENT PROCEDURE WITH NO PARAMETERS;

      IF APA = 0 THEN
      BEGIN COMMENT THIS IS THE AMBIGUITY OF THE PRIMARY;

         IF KIND ¬= "PROCEDURE" THEN
         BEGIN
            IF KIND = "SIMPLE" THEN DAMB(FALSE, PRIM) ELSE
            ERROR(SP, " IS OF WRONG KIND")
         END ELSE
         IF ¬COND THEN DAMB(TRUE, PRIM) ELSE
         ERROR(SP, " IS OF WRONG TYPE")
      END ELSE
      IF APA = 1 THEN
      BEGIN COMMENT THIS IS AMBIGUITY FROM VALUE RIGHT HAND SIDE;

         IF KIND ¬= "PROCEDURE" THEN
         BEGIN
            IF KIND = "SIMPLE" THEN DAMB(FALSE, PRIM) ELSE
            ERROR(SP, " IS OF THE WRONG KIND")
         END ELSE
         IF ¬COND THEN
         BEGIN
            DAMB(TRUE, PRIM);  DAMB(TRUE, VALEXPR)
         END ELSE
         IF TYPE ¬= "REF" THEN DAMB(FALSE, VALEXPR) ELSE
         ERROR(SP, " IS OF THE WRONG TYPE")
      END ELSE
      IF APA = 2 THEN
      BEGIN  COMMENT  THIS IS THE ACTUAL PARAMETER AMBIGUITY;

         IF KIND ¬= "SIMPLE" THEN DAMB(FALSE, AP) ELSE
```

```
         IF ¬COND THEN DAMB(FALSE, PRIM) ELSE
         IF TYPE ¬= "REF" THEN DAMB(FALSE, VALEXPR)
                          ELSE DAMB(FALSE, EXPR)
      END ELSE
      COMMENT THIS IS AN ACTUAL PARAMETER ENCLOSED IN PARETHESIS;

      IF KIND = "LABEL" THEN DAMB(FALSE, EXPR) ELSE
      IF KIND = "SWITCH" OR KIND = "ARRAY" THEN
      ERROR(SP, " IS OF THE WRONG KIND") ELSE
      IF COND THEN
      BEGIN
         IF TYPE = "REF" THEN DAMB(FALSE, EXPR)
                         ELSE DAMB(FALSE, VALEXPR)
      END ELSE
      IF KIND = "SIMPLE" THEN DAMB(FALSE, PRIM) ELSE
      BEGIN DAMB(TRUE, PRIM); DAMB(TRUE, VALEXPR); DAMB(TRUE, EXPR)
      END

   END ELSE
   COMMENT  THIS IS A PROCEDURE WITH PARAMETERS;

   IF KIND ¬= "PROCEDURE" THEN ERROR(SP, " IS OF THE WRONG KIND")
   ELSE
   IF APA = 0 THEN
   BEGIN
      IF COND THEN ERROR(SP, " IS OF THE WRONG TYPE")
   END ELSE
   IF APA = 1 THEN
   BEGIN
      IF ¬COND THEN DAMB(TRUE, VALEXPR) ELSE
      IF TYPE ¬= REF THEN DAMB(FALSE, VALEXPR) ELSE
      ERROR(SP, " IS OF THE WRONG TYPE")
   END ELSE
   COMMENT THIS IS THE AP AMBIGUITY;

   IF ¬COND THEN BEGIN DAMB(TRUE, VALEXPR); DAMB(TRUE, EXPR) END ELSE
   IF TYPE ¬= "REF" THEN DAMB(FALSE, VALEXPR) ELSE DAMB(FALSE, EXPR)
END;


FUNCTION FIXCOND (RULE, SM, SN);
BEGIN COMMENT THIS WILL HANDLE THE ATTACHMENT OF THE LABEL PSEUDO-
         INSTRUCTIONS AND THE GO INSTRUCTION TO THE RULE
         CORRESPONDING TO THE ELSE PART OF A CONDITIONAL EXPRESSION.
         SM STANDS FOR THE LABEL OF THE INSTRUCTION FOLLOWING THE
         CONDITIONAL AND SN FOR THE LABEL OF THE RULE ASSOCIATED
         WITH THE ELSE. THE AUGMENTED RULE IS RETURNED;

   CONS(INSTR ** $(FORMAT := F1;  JLABEL := SM),
     CONS(INSTR ** $(LABELI := SN),
       APEND(RULE, LIST(INSTR ** $(LABELI := SM)))))
END;
```

Reproduced from best available copy.

```
FUNCTION INVDELTA (ENV, E);
BEGIN COMMENT   THIS PROCEDURE MERGES TWO SYMBOL TABLES, ENV
```

```
        REPRESENTING THE GLOBAL ENVIRONMENT AND E THE LOCAL ONE.
        THE ALGOL RENAMING RULES ARE FOLLOWED. THE RESULTING
        TABLE IS RETURNED. NOTICE THAT E IS MODIFIED BUT NOT ENV;

   NEXT1 := FIRST(ENV);
   WHILE ¬NULLB(NEXT1) DO
   BEGIN
      OPER := SELECTOR([NEXT1]);
      IF NULLR(FIND(E, [OPER])) THEN E := *[NEXT1];
      NEXT1 := NEXT([NEXT1])
   END;
   E
END;


PROCEDURE OUTPUT (SN, RULE);
BEGIN  COMMENT THIS PROCEDURE HANDLES THE OUTPUT OF THE LIST RULE.
         ADDITIONALLY IT BINDS LABELS AND ADDRESSES THROUGH A TABLE
         MAP (IMPLEMENTED AS A CONSTRUCT). THE LABELLING AND
         BINDING MECHANISM ARE THE SAME AS THE ONE USED FOR
         TURINGOL. INSTRUCTIONS WITH A COMPONENT JLABEL ARE COPIED
         TO AVOID THE PROBLEM THAT ARISES WHEN AN INSTRUCTION
         BELONGING TO A CLASS SEGMENT IS ALSO PART OF THE CLASS
         SEGMENT OF A CLASS HAVING THE FIRST ONE AS A PREFIX;

   D := 1;  WRITE(/,SN);
   WHILE ¬NULLB(RULE) DO
   BEGIN
      NEXT1 := CAR(RULE);
      IF ¬NULLR(FIND([NEXT1], FORMAT)) THEN
      BEGIN COMMENT THIS IS AN INSTRUCTION SINCE ONLY INSTRUCTIONS
              HAVE A FORMAT COMPONENT. OTHERWISE IT IS A PSEUDO-
              INSTRUCTION. THE INSTRUCTION IS COPIED, THE LABEL IN
              JLABEL IS BOUND TO AN ADDRESS AND THE SEGMENT NUMBER
              IS INCLUDED. THE LABEL IS BOUND IN PARALLEL WITH THE
              PROCEDURE PLACE TO AVOID PASSIVATIONS DUE TO FORWARD
              JUMPS;

         D := D + 1; IF ¬NULLR(FIND([NEXT1], JLABEL)) THEN
         BEGIN
            [NEXT1] := #[NEXT1]; [NEXT1].SN := SN;
            PLACE ([NEXT1], MAP)
         END;
         WRITE(/, [NEXT1])
      END ELSE
      IF ¬NULLR(FIND([NEXT1], LABELI)) THEN
      COMMENT THIS IS A LABEL PSEUDO-INSTRUCTION. UPDATE MAP;

      MAP.[[NEXT1].LABELI] := [D];
      RULE := CDR(RULE)
   END
END;


PROCEDURE PLACE (NEXT1, MAP);
COMMENT   THIS PROCEDURE WILL BIND A LABEL WITH AN ADDRESS IN
```

```
   PARALLEL;
$/ [NEXT1].DISP := MAP.[[NEXT1].JLABEL] /$;


FUNCTION PUTII (RULE, COND);
BEGIN  COMMENT THIS WILL PLACE MARKERS FOR INIT AND INNER AT THE
          BEGINNING AND AT THE END OF A RULE, IF COND IS TRUE. THE
          MARKERS ARE PSEUDO-INSTRUCTIONS WITH A COMPONENT MARK: IT
          IS USED TO PUT MARKERS IN THE RULE OF CLASS BODY WHEN IT IS
          NEITHER A SPLIT BODY OR A BLOCK;

    IF COND THEN CONS(INSTR *= $(MARK := "INIT"),
                      APEND(RULE, LIST(INSTR *= $(MARK := "INNER"))))
            ELSE RULE
END;


PROCEDURE SUBORDINATE (QUALTB, GENUS1, GENUS2);
BEGIN  COMMENT THIS PROCEDURE CHECKS TO SEE IF THE SIMULA
          SUBORDINATION RULES ARE RESPECTED;

   TYPE := GENUS1.TYPE;
   IF TYPE ¬= "U" THEN
   BEGIN
      TYPE1 := GENUS2.TYPE;
      IF TYPE1 ¬= TYPE2 THEN ERROR("SUBORDINATION RULES VIOLATED")
      ELSE
      IF TYPE2 = "REF" THEN
      BEGIN
         ATTR := GENUS1.QUAL;
         NEXT1 := QUALTB.[ATTR]; LEVEL := [NEXT1].LEVEL;
         NEXT2 := QUALTB.[GENUS2.QUAL];
         LEVEL := [NEXT2].LEVEL - LEVEL;
         IF LEVEL < 0 THEN ERROR ("SUBORDINATION RULES VIOLATED")
         ELSE
         WHILE LEVEL > 0 DO
         BEGIN
            NEXT2 := [[NEXT2].PREFIX]; LEVEL := LEVEL - 1
         END;
         IF NEXT1 ¬= NEXT2 THEN
         ERROR("SUBORDINATION RULES VIOLATED")
      END
   END
END;


FUNCTION UNIONDOT (E, E1);
BEGIN COMMENT   THIS FUNCTION WILL JOIN TWO SYMBOL TABLES AND IF
          THERE ARE COMMON NAMES AMONG THE COMPONENTS AN ERROR IS
          NOTED. E CHANGES BUT NOT E1;

   NEXT1 := FIRST(E1);
   WHILE ¬NULLB(NEXT1) DO
   BEGIN
```

```
      OPER := SELECTOR([NEXT1]);
      IF NULLR(FIND(E, [OPER])) THEN E := *[NEXT1] ELSE
      ERROR(OPER, " HAS BEEN DECLARED TWICE");
      NEXT1 := NEXT([NEXT1])
   END;
   E
END;


FUNCTION UNIONR (VIRDECL, VIRDECL1);
BEGIN COMMENT   THIS FUNCTION WILL TAKE TWO CONSTRUCTS AND MERGE THEM
         WITH THE FIRST ONE BEING RETURNED MODIFIED AND THE SECOND
         ONE REMAINING UNCHANGED;

   NEXT1 := FIRST(VIRDECL1);
   WHILE ¬NULL(NEXT1) DO
   BEGIN VIRDECL := *[NEXT1];  NEXT1 := NEXT([NEXT1]) END;
   VIRDECL
END;


FUNCTION UPDQUALTB (QUALTB, CDECL);
BEGIN COMMENT  THIS PROCEDURE WILL UPDATE QUALTB BY INTRODUCING
         ENTRIES CORRESPONDING TO THE CLASSES REPRESENTED IN CDECL.
         EACH ENTRY IN QUALTB CORRESPONDS TO A CLASS, AND IS A
         CONSTRUCT IN WHICH THE COMPONENT PREFIX IS A POINTER TO THE
         QUALTB COMPONENT CORRESPONDING TO THE PREFIX CLASS, CLASSN
         IS A POINTER TO THE SYMBOL TABLE ENTRY FOR THE CLASS, AND
         LEVEL THE NUMBER OF CLASSES IN THE PREFIX SEQUENCE OF THE
         CLASS. CDECL IS A LIST OF POINTERS TO THE SYMBOL TABLE
         ENTRIES OF THE CLASSES DECLARED IN A BLOCK. EACH INSERTION
         IN QUALTB, IS MADE IN PARALLEL (USING THE PROCEDURE AUX)
         TO AVOID ORDERING CDECL. NOTICE THAT THE INSERTIONS CANNOT
         BE MADE SEQUENTIALLY WITHOUT ORDERING CDECL SINCE IF A
         CLASS WERE DECLARED BEFORE ITS PREFIX, THE FUNCTION WOULD
         HANG UP TRYING TO FIND THE PREFIX CLASS AND WOULD NEVER
         DEFINE THE PREFIX;

   WHILE ¬NULLB(CDECL) DO
   BEGIN
      AUX([CAR(CDECL)], QUALTB);  CDECL := CDR(CDECL)
   END;
   QUALTB
END;


FUNCTION VIRMERGE (RULE, VIRDECL);
BEGIN COMMENT THE INITIAL PART OF A RULE IS COMPOSED OF A SEQUENCE
         OF INSTRUCTIONS CORRESPONDING TO THE DECLARATIONS: TO AN
         ARRAY DECLARATION CORRESPONDS A SEQUENCE OF INSTRUCTIONS
         DEFINING THE ARRAY'S BOUNDS FOLLOWED BY A MAK INSTRUCTION
         THAT BUILDS A SEGMENT ASSOCIATED WITH THE ARRAY AND INSERTS
         A REFERENCE TO IT IN THE STACK; TO ANY OTHER DECLARATION
         CORRESPONDS ONE C-INSTRUCTION. IF THE INSTRUCTIONS
         DEFINING THE BOUNDARIES ARE IGNORED, THE N(TH) INSTRUCTION
```

```
        IN THE INITIAL SEQUENCE, PLACES IN THE STACK A WORD WHOSE
        STACK DEPTH IS N. VIRMERGE REPLACES IN RULE THOSE
        INSTRUCTIONS THAT CORRESPOND TO VIRTUAL DECLARATIONS AND
        THAT HAVE BEEN REDEFINED. VIRMERGE IS A CONSTRUCT WHOSE
        COMPONENTS ARE THE NEW INSTRUCTIONS; THE COMPONENTS'
        SELECTORS ARE INTEGERS AND CORRESPOND TO THE SEQUENCE
        NUMBER (IGNORING THE BOUNDARY DEFINITION INSTRUCTIONS) OF
        THE INSTRUCTION TO BE REPLACED. BOUNDARY DEFINITION
        INSTRUCTIONS ARE DELIMITED BY PSEUDO-INSTRUCTIONS WITH
        COMPONENT MARK1;

   NEXT1 := FIRST(VIRDECL); D := SELECTOR([NEXT1]);
   WHILE ¬NULLB(NEXT1) DO
   BEGIN
      RULE1 := RULE;
      WHILE D > 1 DO
      BEGIN
         IF ¬NULLR(FIND(CAR(RULE1), MARK1)) DO
         BEGIN COMMENT THIS IS THE BEGINNING OF A SEQUENCE OF
                  BOUNDARY DEFINITION INSTRUCTIONS. SKIP OVER THE
                  INSTRUCTIONS UNTIL ANOTHER ONE WITH COMPONENT
                  MARK1 IS FOUND;

            RULE1 := CDR(RULE1);
            WHILE NULLR(FIND(CAR(RULE1), MARK1)) DO
            RULE1 := CDR(RULE1);
            RULE1 := CDR(CDR(RULE1))
         END ELSE
         RULE1 := CDR(RULE1);
         D := D - 1
      END;
      CAR(RULE1) := [NEXT1];  NEXT1 := NEXT([NEXT1])
   END;
   RULE
END;


$P1   EXPR ::= VALEXPR

$P2   EXPR ::= REFEXPR

$P3   EXPR ::= DESIGEXPR

      $/PL(EXPR).GENUS.TYPE := "LABEL" /$

$P4   VALEXPR ::= ARITEXPR

$P5   VALEXPR ::= BEXPR

$P7   ARITEXPR ::= SARITEXPR
```

```
$P8   ARITEXPR ::= IFCL SARITEXPR ELSE ARITEXPR

      $/ SN := NEWINTEGER;   SM := NEWINTEGER;
         FJUMP(IFCL) := SN;
         RULE(ARITEXPR) := APEND(RULE(IFCL), APEND(RULE(SARITEXPR),
                                 FIXCOND(RULE(ARITEXPR*), SM, SN)))/$
      $/ PL(ARITEXPR).GENUS.TYPE := COMBTYPE(PL(SARITEXPR),
                                             PL(ARITEXPR*))/$
      $/ APA(SARITEXPR) := IF APA(ARITEXPR)=0 THEN 0 ELSE 4;
         APA(ARITEXPR*) := IF ¬SID(SARITEXPR) THEN 0
                                              ELSE APA(ARITEXPR);
         IF SID(SARITEXPR) AND APA(SARITEXPR)=4 THEN
         BEGIN
            TYPE := PL(ARITEXPR).GENUS.TYPE;
            IF TYPE ¬= "INTEGER" AND TYPE ¬= "REAL" THEN
            ERROR("CONDITIONAL ARITHMETIC EXPRESSION HAS OPERAND OF
TYPE ",TYPE)
         END /$
      $/ SID(ARITEXPR) := SID(SARITEXPR) AND SID(ARITEXPR*) /$


$P9   SARITEXPR ::= TERM


$P10  SARITEXPR ::= AOP TERM

      $/ VALENCE(AOP) := 1;  USE(TERM) := "VALUE";
         RULE(SARITEXPR) := APEND(RULE(TERM), RULE(AOP)) /$
      $/ SID(SARITEXPR) := FALSE /$


$P11  SARITEXPR ::= SARITEXPR AOP TERM

      $/ VALENCE(AOP) := 2; USE(SARITEXPR*) := USE(TERM) := "VALUE";
         APA(SARITEXPR*) := APA(TERM) := 0;
         RULE(SARITEXPR) := APEND(RULE(SARITEXPR*),
                                  APEND(RULE(TERM), RULE(AOP))) /$
      $/ PL(SARITEXPR).GENUS.TYPE := COMBTYPE(PL(SARITEXPR*),
                                              PL(TERM)) /$
      $/ SID(SARITEXPR) := FALSE /$


$P12  AOP ::= +

      $/RULE(AOP) := IF VALENCE(AOP) = 1 THEN NULL ELSE
                     LIST(INSTR *= $(FORMAT := F2; OPER := "+")) /$


$P13  AOP ::= -

      $/RULE(AOP) := LIST(INSTR *= $(FORMAT := F2;
                OPER := IF VALENCE(AOP) = 2 THEN "-" ELSE "NEG")) /$


$P14  TERM ::= FAC


$P15  TERM ::= TERM MOP FAC
```

```
      $/ USE(TERM*) := USE(FAC) := "VALUE"; SID(TERM) := FALSE;
         APA(TERM*) := APA(FAC) := 0;
         RULE(TERM) := APEND(RULE(TERM*),
                             APEND(RULE(FAC), RULE(MOP))) /$
      $/ PL(TERM) := PL(MOP);
         T(MOP) := COMBTYPE(PL(TERM*), PL(FAC)) /$

$P16  MOP ::= *

      $/ RULE(MOP) := LIST(INSTR *= $(FORMAT := F2;  OPER := "*"));
         PL(MOP).GENUS.TYPE := T(MOP) /$

$P17  MOP ::= /

      $/ RULE(MOP) := LIST(INSTR *= $(FORMAT := F2;  OPER := "/"));
         PL(MOP).GENUS.TYPE := "REAL" /$

$P18  MOP ::= DIV
      $/ RULE(MOP):= LIST(INSTR *= $(FORMAT := F2;  OPER := "DIV"));
         PL(MOP).GENUS.TYPE := "INTEGER";
         IF T(MOP) ¬= "INTEGER" THEN
         ERROR("MIXED TYPES IN ""DIV"" OPERATION") /$

$P19  FAC ::= PRIM

$P20  FAC ::= FAC ** PRIM

      $/ PL(FAC).GENUS.TYPE := "REAL"; APA(FAC*) := APA(PRIM) := 0;
         USE (FAC*) := USE(PRIM) := "VALUE"; SID(FAC) := FALSE;
         RULE(FAC) := APEND(RULE(FAC*), APEND(RULE(PRIM),
                   LIST(INSTR *= $(FORMAT := F2;  OPER := "**")))) /$

$P21  PRIM ::= NU

      $/ PL(PRIM).GENUS.TYPE := "INTEGER";  SID(PRIM) := FALSE;
         RULE(PRIM) := LIST(INSTR *= $(FORMAT := F3; V := V(NU))) /$

$P22  PRIM ::= VAR

      $/ SID(PRIM) := TRUE /$
      $/ TYPE := PL(VAR).GENUS.TYPE;
         DISAMV(SP(VAR), DAR(VAR), TYPE ¬= "INTEGER" AND
                TYPE ¬= "REAL", PL(VAR), APA(PRIM)) /$

$P23  PRIM ::= FUNC

      $/ SID(PRIM) := TRUE /$
      $/ TYPE := PL(FUNC).GENUS.TYPE;
         DISAMF(SP(FUNC), TYPE ¬= "INTEGER" AND TYPE ¬= "REAL,
```

```
                PL(FUNC), APA(PRIM), L(FUNC)) /$

$P24  PRIM ::= ( ARITEXPR )

      $/ USE(ARITEXPR) := "VALUE";
         APA(ARITEXPR) := IF APA(PRIM)=2 THEN 3 ELSE APA(PRIM) /$


$P48  VAR ::= ID1

      $/ DAR(VAR) := FALSE /$


$P49  VAR ::= ARID1 [ SUBLIST ]

      $/ DAR(VAR) := TRUE;  USE(ARID1) := "VALUE";
         RULE(VAR) := APEND(RULE(ARID1), APEND(RULE(SUBLIST),
                  LIST(INSTR:= $(FORMAT := F4; USE := USE(VAR))))) /$


$P52  ID1 ::= SIGMA

      $/ NEXT1 := FIND(ENV(ID1), [SP(SIGMA)]);
         IF ¬NULLB(NEXT1) THEN PL(ID1) := [NEXT1] ELSE
         ERROR ("UNDEFINED IDENTIFIER", SP(SIGMA)) /$
      $/ ADDR := CHECKIDENTIFIER(ITEM(ID1), QUALTB(ID1), PL(ID1));
         RULE (ID1) := IF NULLB (ADDR) THEN
                CONS(INSTR := $(FORMAT := F7;  ADDR := ADDR),
                   LIST(INSTR := $(ADDR := PL(ID1).ADDR;
                    IF USE(ID1) THEN FORMAT := F5 ELSE FORMAT := F6)))
                ELSE
                LIST(INSTR := $(ADDR := PL(ID1).ADDR;
                   IF USE(ID1) THEN FORMAT := F7 ELSE FORMAT := F8) /$


$P53  ID1 ::= RID

      $/ IF PL(RID).GENUS.KIND = "CLASS" THEN
         ERROR(SP(RID)," CLASS IDENT. USED IN REMOTE IDENTIFIER") /$


$P83  RID ::= SOBJEXPR . ATTRID

      $/ USE(SOBJEXPR) := "VALUE"; APA(SOBJEXPR) := 0;
         RULE(RID) := APEND(RULE(SOBJEXPR), LIST(INSTR := $(IF
              USE(RID) = "VALUE" THEN FORMAT = F5 ELSE FORMAT := F6;
              ADDR := PL(ATTRID).ADDR))) /$
      $/ PL(RID) := PL(ATTRID) /$
      $/ ENV(ATTRID) := [QUALTB(RID). [PL(SOBJEXPR).GENUS.QUAL].
                                                    CLASSN].LOCALE;
         COMMENT  CHECK TO SEE IF ENV CONTAINS CLASS DECLARATIONS.
            IF YES, REMOTE ACCESS IS ILLEGAL;

         NEXT1 := FIRST(ENV(ATTRID));
         WHILE ¬NULLB(NEXT1) DO
         BEGIN
            IF [NEXT1].GENUS.KIND = "CLASS" THEN
```

```
            ERROR("REMOTE ID. ACCESSES CLASS WHICH HAS CLASSES AS ATT
RIBUTES");  NEXT1 := NEXT([NEXT1]) /$


$P84  ATTRID ::= SIGMA

      $/ NEXT1 := FIND(ENV(ATTRID), [SP(SIGMA)]);
         IF ¬NULLB(NEXT1) THEN PL(ATTRID) := [NEXT1] ELSE
         ERROR("UNDEFINED ATTRIBUTE IDENTIFIER ", SP(SIGMA))/$


$P85  ARID1 ::= ID1

      $/ IF APA(ARID1) = 0 THEN
         BEGIN
            IF PL(ID1).GENUS.KIND ¬= "ARRAY" THEN
            ERROR("ARRAY IDENTIFIER EXPECTED AND NOT FOUND ",
                                                    SP(ID1))
         END /$


$P86  SUBLIST ::= SUBEXPR

      $/ L(SUBLIST) := 1; USE(SUBEXPR) := "VALUE" /$


$P87  SUBLIST ::= SUBLIST , SUBEXPR

      $/ USE(SUBEXPR) := "VALUE"; L(SUBLIST) := L(SUBLIST*) + 1 /$
      $/ RULE(SUBLIST) := APEND(RULE(SUBLIST*), CONS(INSTR *=
                  $(FORMAT := F4;  USE := "VALUE"), RULE(SUBEXPR)) /$


$P88  SUBEXPR ::= ARITEXPR

      $/ APA(ARITEXPR) := 0 /$


$P89  FUNC ::= PROCID1 APPART

      $/ USE(PROCID1) := "VALUE";
         RULE(FUNC) := CONS(INSTR *= $(FORMAT := F11),
           CONS(INSTR *= $(FORMAT := F9;  OPER := "RET"),
            CONS(INSTR *= $(FORMAT := F9;
                           OPER := PL(PROCID1).GENUS.TYPE),
             APEND(RULE(APPART), APEND(RULE(PROCID1),
                           LIST(INSTR *= $(FORMAT := F10)))))))) /$


$P90  PROCID1 ::= ID1

      $/ IF APA = 0 THEN
         BEGIN  IF PL(ID1).GENUS.KIND ¬= "PROCEDURE" THEN
                ERROR("PROCEDURE IDENTIFIER HAS WRONG KIND ",SP(ID1))
         END /$


$P91  APPART ::=
```

```
      $/ RULE(APPART) := NULL;   L(APPART) := 0 /$

$P92  APPART ::= ( APLIST )

$P93  APLIST ::= AP

      $/ L(APLIST) := 1 /$

$P94  APLIST ::= APLIST , AP

      $/ L(APLIST) := L(APLIST*) + 1 /$
      $/ RULE(APLIST) := APEND(RULE(APLIST*), RULE(AP)) /$

$P95  AP ::= EXPR

      $/ SN := NEWINTEGER; USE(EXPR) := "NAME"; APA(EXPR) := 2;
         OUTPUT(APEND(RULE(EXPR),
                      LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE(AP) := BACTUAL(PL(EXPR).GENUS, LL(AP), SN) /$
      $/ LL(EXPR) := LL(AP) + 1 /$

$P96  AP ::= ARID1

      $/SN := NEWINTEGER; USE(ARID1) := "NAME"; APA(ARID1) := 2;
        OUTPUT(APEND(RULE(ARID1),
                     LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE(AP) ::= BACTUAL(PL(ARID1).GENUS, LL(AP), SN) /$
      $/ DAMB(PL(ARID1).GENUS.KIND = "ARRAY", AP) /$

$P97  AP ::= SWID1

      $/ SN := NEWINTEGER; APA(SWID1) := 1;
         OUTPUT(APEND(RULE(SWID1),
                      LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE(AP) := BACTUAL(PL(SWID1).GENUS, LL(AP), SN) /$

$P98  AP ::= PROCID1

      $/ SN := NEWINTEGER; USE(PROCID1) := "NAME"; APA(PROCID1) := 2;
         OUTPUT(APEND(RULE(PROCID1),
                      LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE(AP) := BACTUAL(PL(PROCID1).GENUS, LL(AP), SN) /$
      $/ DAMB(PL(PROCID1).GENUS.KIND = "PROCEDURE", AP) /$

$P103 BEXPR ::= SBOOL

$P104 BEXPR ::= IFCL SBOOL ELSE BEXPR
```

```
      $/ SN := NEWINTEGER; SM := NEWINTEGER; FJUMP(IFCL) := SN;
         PL(BEXPR) := APEND(RULE(IFCL), APEND(RULE(SBOOL),
                                 FIXCOND(RULE(BEXPR*), SM, SN))) /$
      $/ SID(BEXPR) := SID(SBOOL) AND SID(BEXPR) /$
      $/ APA(SBOOL) := IF APA(BEXPR)=0 THEN 0 ELSE 4;
         APA(BEXPR*) := IF ¬SID(SBOOL) THEN 0 ELSE APA(BEXPR);
         COMMENT  THE FOLLOWING TEST IS PERFORMED WHEN THE BEXPR IS
            PART OF AN AMBIGUITY SINCE IN THIS CASE THE TYPE OF SBOOL
            HAS NOT BEEN TESTED. THE CLAUSE (SID(BEXPR*) OR
            ¬SID(BEXPR)) IS IN TO GUARANTEE THAT THE TEST IS
            PERFORMED ONLY AFTER THE AMBIGUTIES IN BEXPR* HAVE BEEN
            ALL RESOLVED;

         IF SID(SBOOL) AND APA(SBOOL)=4 AND (SID(BEXPR*) OR
             ¬SID(BEXPR*)) AND PL(SBOOL).GENUS.TYPE ¬= "BOOLEAN" THEN
         ERROR("CONDITIONAL BOOLEAN EXPRESSION HAS OPERAND WITH TYPE"
                  , PL(SBOOL).GENUS.TYPE) /$


$P105 SBOOL ::= IMPL


$P106 SBOOL ::= SBOOL EQUIV IMPL

      $/ PL(SBOOL) := PL(SBOOL*); APA(SBOOL*) := APA(IMPL) := 0;
         USE(SBOOL*) := USE(IMPL) := "VALUE"; SID(SBOOL) := FALSE;
         RULE(SBOOL) := APEND(RULE(SBOOL*), APEND(RULE(IMPL),
                           LIST(INSTR *= $(FORMAT := F14;
                                           OPER := "EQUIV")))) $/


$P107 IMPL ::= BTERM


$P108 IMPL ::= IMPL IMPLIES BTERM

      $/ PL(IMPL) := PL(IMPL*); USE(IMPL*) := USE(BTERM) := "VALUE";
         APA(IMPL*) := APA(BTERM) := 0; SID(IMPL) := FALSE;
         RULE(IMPL) := APEND(RULE(IMPL*), APEND(RULE(BTERM),
                  LIST(INSTR *= (FORMAT := F14; OPER := "IMPLY")))) /$


$P109 BTERM ::= BFAC


$P110 BTERM ::= BTERM OR BFAC

      $/ PL(BTERM) := PL(BTERM*); APA(BTERM*) := APA(BFAC) := 0;
         USE(BTERM*) := USE(BFAC) := "VALUE"; SID(BTERM) := FALSE;
         RULE(BTERM) := APEND(RULE(BTERM*), APEND(RULE(BFAC),
                  LIST(INSTR *= $(FORMAT := F14; OPER := "OR")))) /$


$P111 BFAC ::= BSEC


$P112 BFAC ::= BFAC AND BSEC
```

```
      $/ PL(BFAC) := PL(BFAC*); USE(BFAC*) := USE(BSEC) := "VALUE";
         APA(BFAC) := APA(BSEC) := 0;  SID(BFAC) := FALSE;
         RULE(BFAC) := APEND(RULE(BFAC*), APEND(RULE(BSEC),
                   LIST(INSTR *= $(FORMAT := F14;  OPER := "AND"))))/$

$P113 BSEC ::= BPRIM

$P114 BSEC ::= ¬ BPRIM

      $/ USE(BPRIM) := "VALUE"; APA(BPRIM) := 0; SID(BSEC) := FALSE;
         RULE(BSEC) := APEND(RULE(BPRIM),
                       LIST(INSTR := $(FORMAT := F14, OPER := "¬"))) /$

$P115 BPRIM ::= LOGVAL

      $/ PL(BPRIM).GENUS := $(KIND := "SIMPLE"; TYPE := "BOOLEAN");
         SID(BPRIM) := FALSE /$

$P116 BPRIM ::= VAR

      $/ SID(BPRIM) := TRUE /$
      $/ DISAMV(SP(VAR), DAR(VAR), PL(VAR).GENUS.TYPE ¬= BOOLEAN,
                                              PL(VAR), APA(BPRIM)) /$

$P117 BPRIM ::= FUNC

      $/ SID(BPRIM) := TRUE /$
      $/ DISAMF(SP(FUNC), PL(FUNC).GENUS.TYPE ¬= "BOOLEAN", PL(FUNC),
                                              APA(BPRIM), L(FUNC)) /$

$P118  BPRIM ::= REL

       $/ USE(REL) := "VALUE";  SID(BPRIM) := FALSE;
          PL(BPRIM).GENUS := $(KIND := "SIMPLE";
                               TYPE := "BOOLEAN ") /$

$P119 BPRIM ::= ( BEXPR )

      $/ USE(BEXPR) := "VALUE";
         APA(BEXPR) := IF APA(BPRIM)=2 THEN 3 ELSE APA(BPRIM) /$

$P120 LOGVAL ::= TRUE

      $/ RULE(LOGVAL) := LIST(INSTR *= $(FORMAT := F15;
                                         OPER := "TRUE")) /$

$P121 LOGVAL ::= FALSE

      $/ RULE(LOGVAL) := LIST( INSTR *= $(FORMAT := F15;
                                          OPER := "FALSE")) /$
```

```
$P122 REL ::= SARITEXPR RELOP SARITEXPR

      $/ APA(SARITEXPR) := APA(SARITEXPR*) := 0;
         RULE(REL) := APEND(RULE(SARITEXPR), APEND(RULE(SARITEXPR*),
                                                   RULE(RELOP))) /$

$P124 REL ::= OBJREL

$P125 REL ::= REFREL

$P126 RELOP ::= <

      $/ RULE(RELOP) := LIST(INSTR *= $(FORMAT := F16;
                                        OPER := "<")) /$

$P127 RELOP ::= <=

      $/ RULE(RELOP) := LIST(INSTR *= $(FORMAT := F16;
                                        OPER := "<=")) /$

$P128 RELOP ::= =

      $/ RULE(RELOP) := LIST(INSTR *= $(FORMAT := F16;
                                        OPER := "=")) /$

$P129 RELOP ::= >=

      $/ RULE(RELOP) := LIST(INSTR *= $(FORMAT := F16;
                                        OPER := ">=")) /$

$P130 RELOP ::= >

      $/ RULE(RELOP) := LIST(INSTR *= $(FORMAT := F16;
                                        OPER := ">")) /$

$P131 RELOP ::= ¬=

      $/ RULE(RELOP) := LIST(INSTR *= $(FORMAT := F16;
                                        OPER := "¬=")) /$

$P133 OBJREL ::= SOBJEXPR IS CLID1

      $/ APA(SOBJEXPR) := 0;
         RULE(OBJREL) := APEND(RULE(SOBJEXPR), APEND(RULE(CLID1),
                   LIST(INSTR *= $(FORMAT := F16; OPER := "IS")))) /$
```

```
$P134 OBJREL ::= SOBJEXPR IN CLID1

      $/ APA(SOBJEXPR) := 0;
         RULE(OBJREL) := APEND(RULE(SOBJEXPR), APEND(RULE(CLID1),
                    LIST(INSTR *= $(FORMAT := F16; OPER := "IN")))) /$


$P135 REFREL ::=  OBJREFREL


$P136 OBJREFREL ::= SOBJEXPR REFCOMP SOBJEXPR

      $/ APA(SOBJEXPR) := APA(SOBJEXPR*) := 0;
         RULE(OBJREFREL) := APEND(RULE(SOBJEXPR),
                              APEND(RULE(SOBJEXPR*), RULE(REFCOMP))) /$


$P137 REFCOMP  ::= ==

      $/ RULE(REFCOMP) := LIST(INSTR *= $(FORMAT := F16;
                                          OPER := "==")) /$


$P138 REFCOMP ::= =/=

      $/ RULE(REFCOMP) := LIST(INSTR *= $(FORMAT := F16;
                                          OPER := "=/=")) /$


$P147 REFEXPR ::= OBJEXPR


$P148 OBJEXPR ::= SOBJEXPR


$P149 OBJEXPR ::= IF BEXPR THEN SOBJEXPR ELSE OBJEXPR

      $/ SM := NEWINTEGER;  SN := NEWINTEGER;  USE(BEXPR) := "VALUE";
         APA(BEXPR) := 0;
         RULE(OBJEXPR) := APEND(RULE(BEXPR), APEND(
                        CONS(INSTR *= $(FORMAT := F17;  JLABEL := SN),
                   RULE(SOBJEXPR)), FIXCOND(RULE(OBJEXPR), SM, SN))) /$
      $/ PL(OBJEXPR).GENUS := $(TYPE := "REF";  QUAL :=
                              CONDQUAL(PL(SOBJEXPR).GENUS.QUAL,
                                       PL(OBJEXPR*).GENUS.QUAL)) /$
      $/ APA(SOBJEXPR) := IF APA(OBJEXPR)=0 THEN 0 ELSE 4;
         APA(OBJEXPR*) := IF ¬SID(OBJEXPR) THEN 0 ELSE APA(OBJEXPR);
         IF PL(SOBJEXPR).GENUS.TYPE ¬= PL(OBJEXPR*).GENUS.TYPE THEN
         ERROR("CONDITIONAL OBJECT EXPRESSION HAS OPERAND OF TYPE ",
                  PL(SOBJEXPR).GENUS.TYPE) /$
      $/ SID(OBJEXPR) := SID(SOBJEXPR) AND SID(OBJEXPR*) /$


$P150 SOBJEXPR ::= NONE

      $/ PL(SOBJEXPR).GENUS := $(TYPE := "REF"; QUAL := -1);
         SID(SOBJEXPR) := FALSE;
         RULE(SOBJEXPR) := LIST(INSTR *= $(FORMAT := F18;
```

```
                                                        QUAL := -1)) /$

$P151 SOBJEXPR ::= VAR

      $/ SID(SOBJEXPR) := TRUE /$
      $/ NEXT1 := PL(VAR).GENUS;
         TYPE := [NEXT1].TYPE;  KIND := [NEXT1].KIND;
         IF APA(SOBJEXPR) = 4 THEN DAMB(KIND ¬= "PROCEDURE") ELSE
         IF ¬DAR(VAR) THEN
         BEGIN
            IF APA(SOBJEXPR) = 3 THEN
            BEGIN
               IF KIND ¬= "SIMPLE" THEN
               BEGIN
                  IF KIND = "LABEL" THEN DAMB(FALSE, EXPR) ELSE
                  IF KIND ¬= "PROCEDURE" THEN
                  ERROR(SP(VAR), " IS OF THE WRONG KIND") ELSE
                  IF COND THEN DAMB(FALSE, EXPR)
                          ELSE DAMB(FALSE, PRIM)
               END ELSE
               IF COND THEN DAMB(EXPR, FALSE) ELSE
               BEGIN  DAMB(TRUE, PRIM); DAMB(TRUE, EXPR) END
            END ELSE
            IF APA(SOBJEXPR) = 2 THEN
            BEGIN
               IF KIND ¬= "SIMPLE" THEN
               BEGIN
                  IF KIND ¬= "LABEL" THEN DAMB(FALSE, AP)
                                     ELSE DAMB(FALSE, EXPR)
               END ELSE
               IF TYPE ¬= "REF" THEN DAMB(FALSE, EXPR) ELSE
               BEGIN
                  DAMB(TRUE, SOBJEXPR); DAMB(TRUE, EXPR);
                  DAMB(TRUE, AP)
               END
            END ELSE
            IF KIND ¬= "SIMPLE" THEN
            BEGIN
               IF KIND = "PROCEDURE" THEN DAMB(FALSE, SOBJEXPR) ELSE
               ERROR(SP(VAR)," IS OF THE WRONG KIND")
            END ELSE
            IF TYPE = "REF" THEN DAMB(TRUE, SOBJEXPR) ELSE
            ERROR(SP(VAR), " IS OF THE WRONG TYPE")
         END ELSE
         IF APA(SOBJEXPR) = 2 OR APA(SOBJEXPR) = 3 THEN
         BEGIN
            IF KIND ¬= "ARRAY" THEN
            BEGIN
               IF KIND = "SWITCH" THEN DAMB(FALSE, EXPR) ELSE
               ERROR(SP(VAR), " IS OF THE WRONG KIND")
            END ELSE
            DAMB(TYPE = "REF", EXPR)
         END ELSE
         IF TYPE ¬= "REF" OR KIND ¬= "ARRAY" THEN
         ERROR(SP(VAR), " HAS WRONG TYPE OR KIND) /$
```

```
$P152 SOBJEXPR ::= FUNC

      $/ SID(SOBJEXPR) := TRUE /$
      $/ NEXT1 := PL(FUNC).GENUS;
         TYPE := [NEXT1].TYPE; KIND := [NEXT1].KIND;
         IF APA(SOBJEXPR) = 4 THEN DAMB(KIND ¬= "SIMPLE") ELSE
         IF L(FUNC) = 0 THEN
         BEGIN
            IF APA(SOBJEXPR) = 3 THEN
            BEGIN
               IF KIND ¬= "PROCEDURE" THEN
               BEGIN
                  IF KIND ="LABEL" THEN DAMB(FALSE, EXPR) ELSE
                  IF KIND ¬= "SIMPLE" THEN
                  ERROR(SP(SOBJEXPR), " IS OF THE WRONG KIND") ELSE
                  IF COND THEN DAMB(FALSE, EXPR)
                          ELSE DAMB(FALSE, PRIM)
               END ELSE
               IF COND THEN DAMB(FALSE, EXPR) ELSE
               BEGIN DAMB(TRUE, PRIM);  DAMB(TRUE, EXPR)  END
            END ELSE
            IF APA(SOBJEXPR) = 2 THEN
            BEGIN
               IF KIND = "PROCEDURE" THEN DAMB(FALSE, AP)
                                     ELSE DAMB(FALSE, SOBJEXPR)
            END ELSE
            IF TYPE ¬= "REF" THEN
            ERROR(SP(FUNC), " IS OF THE WRONG TYPE") ELSE
            IF KIND = "PROCEDURE" THEN DAMB(TRUE, SOBJEXPR) ELSE
            IF KIND = "SIMPLE" THEN DAMB(FALSE, SOBJEXPR) ELSE
            ERROR(SP(FUNC), " IS OF THE WRONG KIND")
         END ELSE
         IF APA(SOBJEXPR) = 2 THEN
         BEGIN
            IF KIND = "PROCEDURE" THEN DAMB(TYPE = "REF", EXPR) ELSE
            ERROR(SP(FUNC), " IS OF THE WRONG KIND")
         END ELSE
         IF KIND ¬= "PROCEDURE" OR TYPE ¬= "REF" THEN
         ERROR(SP(FUNC), " IS OF THE WRONG KIND") /$


$P153 SOBJEXPR ::= OBJGEN
      $/ SID(SOBJEXPR) := FALSE /$


$P154 SOBJEXPR ::= LOCOBJ
      $/ SID(SOBJEXPR) := FALSE /$


$P155 SOBJEXPR ::= QUALOBJ
      $/ SID(SOBJEXPR) := FALSE /$


$P156 SOBJEXPR ::= ( OBJEXPR )

      $/ USE(OBJEXPR) := "VALUE";
         APA(OBJEXPR) := IF APA(SOBJEXPR)=2 THEN 3
                                            ELSE APA(SOBJEXPR) /$
```

```
$P157 OBJGEN ::= NEW CLID2 APPART

      $/ RULE(OBJGEN) := APEND(CONS(INSTR *= $(FORMAT := F11),
                       RULE(APPART)), APEND(RULE(CLID2),
                     CONS(INSTR *= $(FORMAT := F19;
                                     NFORMALS := PL(CLID2).NFORMALS),
                          LIST(INSTR *= $(FORMAT := F12))))) /$
      $/ PL(OBJGEN).GENUS := $(TYPE := "REF";
                               QUAL := PL(CLID2).SEGMENT) /$
      $/ IF PL(CLID2).NFORMALS ¬= L(APPART) THEN
         ERROR("WRONG NUMBER OF PARAMETERS IN CLASS ", SP(CLID2)) /$


$P157A CLID2 ::= ID1

      $/ IF PL(ID1).GENUS.KIND ¬= "CLASS" THEN
         ERROR(SP(ID1), " NOT A CLASS IDENTIFIER") /$


$P158 CLID1 ::= SIGMA

      $/ NEXT1 := FIND(ENV(CLID1), [SP(SIGMA)]);
         IF ¬NULLB(NEXT1) THEN PL(CLID1) := [NEXT1] ELSE
         ERROR(SP(SIGMA), " UNDECLARED CLASS IDENTIFIER") /$
      $/ IF PL(CLID1).GENUS.KIND ¬= "CLASS" THEN
         ERROR(SP(SIGMA), " NOT A CLASS IDENTIFIER") /$


$P159 LOCOBJ ::= THIS CLID1

      $/ RULE(LOCOBJ) := LIST(INSTR *= $(FORMAT := F18;
                        OPER := "THIS"; QUAL := PL(CLID1).SEGMENT)) /$
      $/ PL(LOCOBJ).GENUS := (TYPE := "REF"; KIND := "SIMPLE";
                               QUAL := PL(CLID1).SEGMENT) /$


$P160 QUALOBJ ::= SOBJEXPR QUA CLID1

      $/ APA(SOBJEXPR) := 0;
         PL(QUALOBJ).GENUS := $(TYPE := "REF"; KIND := "SIMPLE";
                                 QUAL := PL(CLID1).SEGMENT) /$


$P161 DESIGEXPR ::= SDESIGEXPR


$P162 DESIGEXPR ::= IFCL SDESIGEXPR ELSE DESIGEXPR

      $/ SM := NEWINTEGER; SN := NEWINTEGER; FJUMP(IFCL) := SN
         RULE(DESIGEXPR) := APEND(RULE(IFCL), APEND(RULE(SDESIGEXPR),
                                  FIXCOND(RULE(DESIGEXPR*), SM, SN))) /$
      $/ APA(SDESIGEXPR) := IF APA(SDESIGEXPR)=0 THEN 0 ELSE 4;
         IF PL(SDESIGEXPR).GENUS.TYPE ¬= PL(DESIGEXPR*).GENUS.TYPE
         THEN
         ERROR("CONDITIONAL DESIGNATIONAL EXPRESSION HAS OPERAND OF T
YPE ", PL(SDESIGEXPR).GENUS.TYPE) /$
```

```
$P163 SDESIGEXPR ::= LABEL1

$P164 SDESIGEXPR ::= SWDESIG

$P165 SDESIGEXPR ::= ( DESIGEXPR )
      $/ APA(DESIGEXPR) := IF APA(SDESIGEXPR)=2 THEN 3
                                          ELSE APA(SDESIGEXPR) /$

$P166 LABEL1 ::= SIGMA

      $/ RULE(LABEL1) := LIST(INSTR *= $(FORMAT := F7;
                                          ADDR := PL(LABEL1).ADDR)) /$
      $/ NEXT1 := FIND(ENV(LABEL1), [SP(SIGMA)]);
         IF ¬NULLB(NEXT1) THEN PL(LABEL1) := [NEXT1] ELSE
         ERROR(SP(SIGMA), " UNDECLARED LABEL") /$
      $/ KIND := PL(LABEL1).GENUS.KIND;
         IF APA(LABEL1) = 3 THEN DAMB(KIND = "LABEL", EXPR) ELSE
         IF APA(LABEL1) = 2 THEN
         BEGIN
            IF KIND ¬= "LABEL" THEN
            BEGIN
               IF KIND ¬= "SIMPLE" THEN DAMB(FALSE,AP)
                                   ELSE DAMB(FALSE, EXPR)
            END ELSE
            BEGIN DAMB(TRUE, EXPR);  DAMB(TRUE, AP)  END
         END ELSE
         IF APA(LABEL1)=0 AND KIND ¬= "LABEL" THEN
         ERROR(SP(SIGMA), " NOT A LABEL") /$

$P167 SWDESIG ::= SWID1 [ SUBEXPR ]

      $/ USE(SUBEXPR) := "VALUE";
         RULE(SWDESIG) := APEND(RULE(SWID1), APEND(RULE(SUBEXPR),
                  LIST(INSTR *= $(FORMAT := F4; USE := VALUE)))) /$
      $/ PL(SWDESIG) := IF PL(SWID1).GENUS.KIND = "SWITCH" THEN
                        $(GENUS.KIND := "LABEL") ELSE PL(SWID1) /$

$P168 SWID1 ::= SIGMA

      $/ RULE(SWID1) := LIST(INSTR *= $(FORMAT := F7;
                                          ADDR := PL(SWID1).ADDR)) /$
      $/ NEXT1 := FIND(ENV(SWID1), [SP(SIGMA)]);
         IF ¬NULLB(NEXT1) THEN PL(SWID1) := [NEXT1] ELSE
         ERROR(SP(SIGMA), " UNDECLARED SWITCH IDENTIFIER") /$
      $/ KIND := PL(SWID1).GENUS.KIND;
         IF APA(SWID1) = 2 OR APA(SWID1) = 3 THEN
         BEGIN
            IF KIND = "SWITCH" THEN DAMB(TRUE, EXPR) ELSE
            IF KIND = "ARRAY" THEN DAMB(FALSE,EXPR) ELSE
            ERROR(SP(SIGMA), " IS OF THE WRONG KIND")
         END ELSE
```

```
      IF APA(SWID1) = 1 THEN DAMB(KIND = "SWITCH", AP) ELSE
      IF APA(SWID1) = 0 AND KIND ¬= "SWITCH" THEN
      ERROR(SP(SIGMA), "NOT A SWITCH IDENTIFIER") /$


$P169 PROGRAM ::= BLOCK

      $/ SN := NEWINTEGER; LL(BLOCK) := 4; DO(BLOCK) := 1 ;
         CL(BLOCK) := 0; ITEM(BLOCK) := NULL;  ENV1(BLOCK) := NULL;
         ENVA(BLOCK) := NULL; QUALTB(BLOCK) := NULL;
         BEGUN(BLOCK) := TRUE; ENV(BLOCK) := E(BLOCK);
         WRITE ("ORIGIN =", SN, /);
         OUTPUT(APEND(UNDECL(BLOCK), APEND(RULE(BLOCK),
                              LIST(INSTR *= $(FORMAT := F12)))), SN) /$


$P170 PROGRAM ::= COMPST

      $/ SN := NEWINTEGER; LL(COMPST) := 4;  DO(COMPST) := 1;
         CL(COMPST) := 0; ITEM(COMPST) := NULL; ENV1(COMPST) := NULL;
         ENV(COMPST) := E(COMPST); QUALTB(COMPST) := NULL;
         WRITE( "ORIGIN =",SN, /);
         OUTPUT(APEND(UNDECL(COMPST).APEND(RULE(COMPST),
                              LIST(INSTR *= $(FORMAT := F12)))), SN) /$


$P171 COMPST ::= UNLCOMP


$P172 COMPST ::= LABEL0 :  COMPST

      $/ DN := CHECKVIRT(ENV1(COMPST), SP(LABEL0), "LABEL") /$
      $/ SN := NEWINTEGER;
         RULE(COMPST) := CONS(INSTR *= $(LABEL1 := SN),
                                                RULE(COMPST*)) /$
      $/ INSTR := $(FORMAT := F20; JLABEL := SN) /$
      $/ UNDECL(COMPST) :=  IF DN ¬= 0 THEN UNDECL(COMPST*) ELSE
                                    CONS(INSTR, UNDECL(COMPST*)) /$
      $/ VIRDECL(COMPST) := IF DN = 0 THEN VIRDECL(COMPST*) ELSE
                              PUTIN(VIRDECL(COMPST*) : [DN] := INSTR) /$
      $/ D(COMPST) := IF DN = 0 TEN D(COMPST*) +1 ELSE D(COMPST*) /$
      $/ DO(COMPST*) := IF DN = 0 THEN DO(COMPST) + 1
                                    ELSE DO(COMPST) /$
      $/ E(COMPST) := IF DN ¬= 0 THEN E(COMPST*) ELSE
                UNIONDOT(E(COMPST*), E1 *= $([SP(LABEL0)] :=
                  $(GENUS := $(KIND := "LABEL"; TYPE := "LABEL");
                     ADDR := $(FORMAT := F21; LN := LL(COMPST);
                                                DN := DO(COMPST)))))) /$


$P172A LABEL0 ::= SIGMA


$P173 UNLCOMP ::= BEGIN COMPT

      $/ IF FIRSTST(COMPT) THEN DAMB(TRUE, 1) /$
```

```
$P174 BLOCK ::= UNLBLOCK

      $/ NOLABEL(BLOCK) := TRUE /$


$P175 BLOCK ::= LABEL0 : BLOCK

      $/ SM := NEWINTEGER; SN := NEWINTEGER; NOLABEL(BLOCK) := FALSE;
         INSTR := $(FORMAT := F20; JLABEL := SM);
         COND := BEGUN(BLOCK) AND NOLABEL(BLOCK*) /$
      $/ DN := CHECKVIRT(ENV1(BLOCK), SP(LABEL0), "LABEL") /$
      $/ RULE(BLOCK) := CONS(INSTR *= $(LABEL1 := SM),
                             IF ¬COND THEN RULE(BLOCK*)ELSE
                             CONS(INSTR *= $(FORMAT := F11),
                                  LIST(INSTR *= $(FORMAT := F22;
                                       SN := SN; LEVEL := LL(BLOCK) + 1))) /$
      $/ IF COND THEN OUTPUT(APEND(RULE(BLOCK*),
                                   LIST(INSTR := $(FORMAT := F12))), SN) /$
      $/ D0(BLOCK*) := IF COND THEN 1 ELSE IF DN = 0 THEN
                                           D0(BLOCK) + 1 ELSE D0(BLOCK) /$
      $/ CL(BLOCK*) := IF COND THEN 0 ELSE CL(BLOCK);
         ENV1(BLOCK*) := IF COND THEN NULL ELSE ENV1(BLOCK);
         ENVA(BLOCK*) := IF COND THEN NULL ELSE ENVA(BLOCK);
         E(BLOCK) := IF COND THEN E1 ELSE UNIONDOT(E(BLOCK*),E1) /$
      $/ LL(BLOCK*) := IF COND THEN LL(BLOCK) + 1 ELSE LL(BLOCK) /$
      $/ D(BLOCK) := IF DN = 0 THEN D(BLOCK*) + 1 ELSE D(BLOCK*) /$
      $/ UNDECL(BLOCK) := IF DN = 0 THEN CONS(INSTR, UNDECL(BLOCK*)
                                      ELSE UNDECL(BLOCK*) /$
      $/ VIRDECL(BLOCK) := IF DN = 0 THEN VIRDECL(BLOCK*) ELSE
                           PUTIN(VIRDECL(BLOCK*) : [DN] := INSTR) /$
      $/ E1 := IF DN ¬= 0 THEN NULL ELSE $([SP(LABEL0)] :=
               $(GENUS := $(KIND := "LABEL"; TYPE := "LABEL");
                  ADDR := $(FORMAT := F21; LN := LL(BLOCK);
                                           DN := D0(BLOCK)))) '
      $/ ENV(BLOCK*) := IF ¬= COND THEN ENV(BLOCK) ELSE
                                INVDELTA(ENV(BLOCK), E(BLOCK*)) /$


$P176 BLOCK ::= UNLPREBLOCK

      $/ NOLABEL(BLOCK) := TRUE /$


$P179 UNLBLOCK ::= BLOCKHEAD ; COMPT

      $/ SN := NEWINTEGER;
         IF ¬BEGUN(UNLBLOCK) THEN OUTPUT(APEND(RULE1,
                                   LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE1 := APEND(RULE(BLOCKHEAD), APEND(UNDECL(COMPT),
                                         PUTII(RULE(COMPT), TRUE))) /$
      $/ RULE(UNLBLOCK) := IF BEGUN(UNLBLOCK) THEN RULE1 ELSE
               CONS(INSTR *= $(FORMAT := F11), LIST(INSTR *=
                     $(FORMAT := F22; LL := LL(COMPT); SN := SN))) /$
      $/ E(UNLBLOCK) := IF ¬BEGUN(UNLBLOCK) THEN NULL ELSE
                                   UNIONDOT(E(BLOCKHEAD), E(COMPT)) /$
      $/ D(UNLBLOCK) := IF ¬BEGUN(UNLBLOCK) THEN 0 ELSE
                                          D(BLOCKHEAD) + D(COMPT) /$
      $/ D0(BLOCKHEAD) := IF ¬BEGUN(UNLBLOCK) THEN 1
```

```
                                                    ELSE DO(UNLBLOCK);
      DO(COMPT) := DO(BLOCKHEAD) + D(BLOCKHEAD) /$
   $/ LL(BLOCKHEAD) := LL(COMPT) := IF BEGUN(BLOCKHEAD) THEN
                                 LL(UNLBLOCK)ELSE LL(UNLBLOCK) +1 /$
   $/ UNDECL(UNLBLOCK) := NULL;
      ENVA(BLOCKHEAD) := IF BEGUN(UNLBLOCK) THEN ENVA(UNLBLOCK)
                                                    ELSE ENV(UNLBLOCK);
      ENV1(BLOCKHEAD) := ENV1(COMPT) := IF ¬BEGUN(UNLBLOCK) THEN
                                          NULL ELSE ENV1(UNLBLOCK);
      ENV(BLOCKHEAD) := ENV(COMPT) := IF BEGUN(UNLBLOCK) THEN
                        ENV(UNLBLOCK) ELSE INVDELTA(ENV(UNLBLOCK),
                                 UNIONDOT(E(BLOCKHEAD),E(COMPT))) /$
$/ VIRDECL(UNLBLOCK) := IF ¬BEGUN(UNLBLOCK) THEN NULL ELSE
                      UNIONR(VIRDECL(BLOCKHEAD), VIRDECL(COMPT)) /$
$/ CL(BLOCKHEAD) := IF ¬BEGUN(UNLBLOCK) THEN 0 ELSE
                                                    CL(UNLBLOCK) /$
$/ QUALTB(BLOCKHEAD) := QUALTB(COMPT) :=
                   UPDQUALTB(QUALTB(UNLBLOCK), CDECL(BLOCKHEAD)) /$
$/ IF AEMDEC(BLOCKHEAD) THEN
   BEGIN
      IF NUMDEC(BLOCKHEAD) = 1 THEN DAMB(FALSE, 1) ELSE
                                    DAMB(FALSE, 2)
   END ELSE
   IF EMDEC(BLOCKHEAD) THEN DAMB(FALSE, 1) ELSE
   IF FIRSTST(COMPT) THEN DAMB(TRUE, 1) /$


$P180 UNLPREBLOCK ::= BLOCKPRE MBLOCK

      $/ UNDECL(UNPREBLOCK) := NULL; VIRDECL(UNLPREBLOCK) := NULL;
         D(UNLPREBLOCK) := 0; CL(MBLOCK) := SN;
         BEGUN(MBLOCK) := TRUE; E(UNLPREBLOCK) := NULL;
         SN := NEWINTEGER; SM := NEWINTEGER;
         OUTPUT(LIST(INSTR *= $(FORMAT := F31;  SN := SN;  SM := SM;
                                OBJECT := PL(BLOCKPRE).OBJECT), SM);
         OUTPUT(VIRMERGE(CONCATENATE(PL(BLOCKPRE).CODE,
                             APEND(UNDECL(MBLOCK), RULE(MBLOCK))),
                                                VIRDECL(MBLOCK)),SN) /$
      $/ RULE(UNLPREBLOCK) := CONS(INSTR *= $(FORMAT := F11),
               APEND(RULE(BLOCKPRE), CONS(INSTR *= $(FORMAT := F32;
                SN := SM;  LL := LL(UNLPREBLOCK) + 1), CONS(INSTR *=
                 $(FORMAT := F19; NFORMALS := PL(BLOCKPRE).NFORMALS),
                  CONS(INSTR *= $(FORMAT := F12),
                   LIST(INSTR *= (FORMAT := F27))))))) /$
      $/ ENV(MBLOCK) := INVDELTA(ENV(UNLPREBLOCK),
                         INVDELTA(PL(BLOCKPRE).LOCALE,
                          INVDELTA(ENV1(MBLOCK), E(MBLOCK)))) /$
      $/ ENV1(MBLOCK) := #PL(BLOCKPRE).VIRTUALE /$
      $/ ENVA(MBLOCK) := INVDELTA(ENV(MBLOCK),
                                  E1 *= PL(BLOCKPRE).FORMALE) /$
      $/ DO(MBLOCK) := PL(BLOCKPRE).NLOCALS + 1 /$
      $/ LL(MBLOCK) := LL(UNLPREBLOCK) + 2 /$


$P181 BLOCKPRE ::= CLID1 APPART

      $/ RULE(BLOCKPRE) := RULE(APPART);
```

```
      IF LL(BLOCKPRE) ¬= PL(CLID1).ADDR.LN THEN
      ERROR("BLOCK PREFIX IS NOT AT THE SAME LEVEL AS BLOCK");
      IF L(APPART) ¬= PL(CLID1).NFORMALS THEN
      ERROR("DIFFERENT NUMBER OF FORMAL AND ACTUAL PARAMETERS") /$

$P182 MBLOCK ::= UNLBLOCK

$P183 MBLOCK ::= UNLCOMP

      $/ RULE(MBLOCK) := PUTTI(RULE(UNLCOMP), TRUE) /$

$P184 BLOCKHEAD ::= BEGIN DECL

      $/ NUMDEC(BLOCKHEAD) := 1; AEMDEC(BLOCKHEAD) := EMDEC(DECL) /$

$P185 BLOCKHEAD ::= BLOCKHEAD ; DECL

      $/ RULE(BLOCKHEAD) := APEND(RULE(BLOCKHEAD*), RULE(DECL)) /$
      $/ VIRDECL(BLOCKHEAD) := UNIONR(VIRDECL(BLOCKHEAD*),
                                      VIRDECL(DECL)) /$
      $/ E(BLOCKHEAD) := UNIONDOT(E(BLOCKHEAD*), E(DECL)) /$
      $/ D(BLOCKHEAD) := D(BLOCKHEAD*) + D(DECL) /$
      $/ NUMDEC(BLOCKHEAD) := NUMDEC(BLOCKHEAD) + 1 /$
      $/ AEMDEC(BLOCKHEAD) := AEMDEC(BLOCKHEAD*) AND AEMDEC(DECL) /$
      $/ DO(DECL) := DO(BLOCKHEAD) + D(BLOCKHEAD) /$
      $/ CDECL(BLOCKHEAD) := APEND(CDECL(BLOCKHEAD*), CDECL(DECL)) /$

$P186 DECL ::= TYPEDECL

      $/ CDECL(DECL) := NULL; VIRDECL(DECL) := NULL;
         EMDEC(DECL) := FALSE /$

$P187 DECL ::= ARDECL

      $/ CDECL(DECL) := NULL;  VIRDECL(DECL) := NULL;
         EMDEC(DECL) := FALSE /$

$P188 DECL ::= SWDECL

      $/ CDECL(DECL) := NULL; EMDEC(DECL) := FALSE /$

$P189 DECL ::= PROCDECL

      $/ CDECL(DECL) := NULL;  EMDEC(DECL) := FALSE /$

$P190 DECL ::= CLDECL

      $/ EMDEC(DECL) := FALSE, VIRDECL(DECL) := NULL;
         CDECL(DECL) := LIST(FIRST(E(CLDECL))) /$
```

```
$P191 DECL ::=

      $/ RULE(DECL) := NULL;  E(DECL) := NULL; VIRDECL(DECL) := NULL;
         EMDEC(DECL) := TRUE;  CDECL(DECL) := NULL; D(DECL) := 0 /$


$P192 TYPEDECL ::= TYPEN  TYPELIST

      $/ INSTR := IF NULLR(FIND(TYPDS(TYPEN),QUAL)
                  THEN $(FORMAT := F9;  OPER := TYPDS(TYPEN).TYPE)
                  ELSE $(FORMAT := F18;  QUAL := TYPDS(TYPEN).TYPE)
         DN := D(TYPELIST);  RULE1 := NULL;
         WHILE DN > 0 DO
         BEGIN  RULE1 := CONS(INSTR, RULE1); DN := DN - 1  END /$
      $/ TYPD(TYPELIST) := TYPDS(TYPEN) /$


$P193 TYPEN ::= VALTYPE


$P194 TYPEN ::= REFTYPE


$P195 VALTYPE ::= REAL

      $/ TYPDS := $(KIND := "SIMPLE"; TYPE := "REAL") /$


$P196 VALTYPE ::= INTEGER

      $/ TYPDS := $(KIND := "SIMPLE"; TYPE := "INTEGER") /$


$P197 VALTYPE ::= BOOLEAN

     $/ TYPDS := $(KIND := "SIMPLE"; TYPE := "BOOLEAN") /$


$P199 REFTYPE ::= OBJREF


$P200 OBJREF ::= REF ( QUALIF )

      $/ TYPDS := $(KIND := "SIMPLE"; TYPE := "REF";
                                  QUAL := PL(QUALIF).SEGMENT) /$


$P201 QUALIF ::= SIGMA

      $/ NEXT1 := FIND(ENV(QUALIF), [SP(SIGMA)]);
         IF ¬NULLB(NEXT1) THEN PL(QUALIF) := [NEXT1] ELSE
         ERROR(SP(SIGMA), " UNDECLARED CLASS IDENTIFIER");
         IF PL(QUALIF).GENUS.KIND ¬= "CLASS" THEN
         ERROR(SP(SIGMA), " NOT A CLASS IDENTIFIER") /$
```

```
$P202 TYPELIST ::= SIGMA

      $/ D(TYPELIST) := 1;
         E(TYPELIST).[SP(SIGMA)] := $(GENUS := TYPD(TYPELIST);
                        ATTR := CL(TYPELIST); ADDR := $(FORMAT := F21;
                           DN := DO(TYPELIST); LN := LL(TYPELIST))) /$


$P203 TYPELIST ::= SIGMA , TYPELIST

      $/ E(TYPELIST) := UNIONDOT(E(TYPELIST*), E1 *= ([SP(SIGMA)] :=
               $(GENUS := TYPDS(TYPELIST); ATTR := CL(TYPELIST);
                  ADDR := $(FORMAT := F21; DN := DO(TYPELIST);
                              LN := LL(TYPELIST)))) /$
      $/ D(TYPELIST) := D(TYPELIST*) + 1 /$
      $/ DO(TYPELIST*) := DO(TYPELIST) + 1 /$


$P206 ARDECL ::= ARRAY ARLIST

      $/ TYPD(ARLIST) := $(KIND := "ARRAY";  TYPE := "REAL") /$


$P207 ARDECL ::= TYPEN ARRAY LIST

      $/ TYPD(ARLIST) := PUTIN(TYPDS(TYPEN) : KIND := "ARRAY") /$


$P208 ARLIST ::= ARSEG

      $/ RULE(ARLIST) := APEND(RULE(ARSEG), LIST(INSTR *=
                           $(FORMAT := F23;  GENUS := TYPD(ARSEG);
                                D := D(ARSEG)-1;  L := L(ARSEG)))) /$


$P209 ARLIST ::= ARLIST , ARSEG

      $/ RULE(ARLIST) := APEND(RULE(ARLIST*), APEND(RULE(ARSEG),
               LIST(INSTR *= $(FORMAT := F23; L := L(ARSEG);
                        D := D(ARSEG)-1;  GENUS := TYPD(ARLIST))))) /$
      $/ E(ARLIST) := UNIONDOT(E(ARLIST*), E(ARSEG)) /$
      $/ D(ARLIST) := D(ARLIST*) + D(ARSEG) /$
      $/ DO(ARSEG) := DO(ARLIST) + D(ARLIST*) /$


$P210 ARSEG ::= ARID [ BOUNDPLIST ]

      $/ RULE(ARSEG) := CONS(INSTR *= $(MARK1 := "IGNORE"),
                     APEND(RULE(BOUNDPLIST),
                        LIST(INSTR *= $(MARK1 := "END IGNORE")))) /$
      $/ D(ARSEG) := 1;
         E(ARSEG).[SP(ARID)] := $(GENUS := TYPD(ARSEG);
              ATTR := CL(ARSEG);  N:= L(BOUNDPLIST); ADDR :=
               $(FORMAT := F21; DN := DO(ARSEG); LN := LL(ARSEG))) /$


$P211 ARSEG := ARID , ARSEG
```

```
      $/ E(ARSEG) := UNIONDOT(E(ARSEG*), E1 *= $([SP(ARID] :=
                     $(GENUS := TYPD(ARSEG);  N := L(ARSEG*);
                        ATTR := CL(ARSEG); ADDR := $(FORMAT := F21;
                              DN := D0(ARSEG); LN := LL(ARSEG))))) /$
      $/ D(ARSEG) := D(ARSEG*) + 1 /$
      $/ D0(ARSEG*) := D0(ARSEG) + 1 /$


$P212 ARID ::= SIGMA


$P213 BOUNDPLIST ::= BOUNDP

      $/ L(BOUNDPLIST) := 1 /$


$P214 BOUNDPLIST ::= BOUNDPLIST , BOUNDP

      $/ RULE(BOUNDPLIST) := APEND(RULE(BOUNDPLIST*), RULE(BOUNDP))/$
      $/ L(BOUNDPLIST) := L(BOUNDPLIST*) + 1 /$


$P215 BOUNDP ::= BOUND : BOUND

      $/ RULE(BOUNDP) := APEND(RULE(BOUND), RULE(BOUND*)) /$


$P216 BOUND ::= ARITEXPR

      $/ APA(ARITEXPR) := 0; USE(ARITEXPR) := "VALUE";
         ENV(ARITEXPR) := ENVA(BOUND) /$


$P218 SWDECL ::= SWITCH SWID :=  SWLIST

      $/ SN := NEWINTEGER;
         INSTR := $(FORMAT := F24; SN := SN;  L := L(SWLIST)) /$
      $/ DN := CHECKVIRT(ENV1(SWDECL), SP(SWID), "SWITCH") /$
      $/ OUTPUT(RULE(SWLIST), SN) /$
      $/ RULE(SWDECL) := IF DN ¬= 0 THEN NULL ELSE LIST(INSTR) /$
      $/ VIRDECL(SWDECL) := IF DN = 0 THEN NULL ELSE
                                              $([DN] := INSTR) /$
      $/ D(SWDECL) := IF DN = 0 THEN 1 ELSE 0 /$
      $/ E1.[SP(SWID)] := $(GENUS := $(KIND := "SWITCH";
                           TYPE := "SWITCH");  N := L(SWLIST);
                     ATTR := CL(SWDECL);  ADDR := $(FORMAT := F21;
                          LN := LL(SWDECL);
                           DN := IF DN = 0 THEN D0(SWDECL) ELSE DN));
         E(SWDECL) := IF DN = 0 THEN E1 ELSE
         BEGIN
            %PUTIN(ENV(SWDECL): [SP(SWID)] := FIRST(E1)); NULL
         END /$


$P218A SWID ::= SIGMA


$P219 SWLIST ::= DESIGEXPR
```

```
      $/ SN := NEWINTEGER;  L(SWLIST) := 1;  APA(DESIGEXPR) := 0;
         OUTPUT(APEND(RULE(DESIGEXPR),
                      LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE(SWLIST) := LIST(INSTR *= $(FORMAT := F13; SN := SN;
                         LL := LL(SWLIST) + 1 ; TYPE := "LABEL")) /$


$P220 SWLIST ::= SWLIST , DESIGEXPR

      $/ SN := NEWINTEGER;  APA(DESIGEXPR) := 0;
         OUTPUT(APEND(RULE(DESIGEXPR),
                      LIST(INSTR *= $(FORMAT := F12))), SN) /$
      $/ RULE(SWLIST) := APEND(RULE(SWLIST*), LIST(INSTR *=
                               $(FORMAT := F13; LL := LL(SWLIST) + 1;
                                 TYPE := "LABEL"; SN := SN))) /$
      $/ L(SWLIST) := L(SWLIST*) + 1 /$


$P221 PROCDECL ::= TYPEP PROCEDURE PROCHEAD PROCBODY

      $/ SN := NEWINTEGER;
         INSTR := $(FORMAT := F33; SN := SN; LL := LL(PROCBODY));
         DN := CHECKVIRT(ENV1(PROCDECL), SP(PROCHEAD), "PROCEDURE")/$
      $/ OUTPUT(APEND(RULE(PROCHEAD), APEND(UNDECL(PROCBODY),
                      LIST(INSTR *= $(FORMAT := F12))))), SN)) /$
      $/ ENVA(PROCBODY) := INVDELTA(ENV(PRODECL), E(PROCHEAD)) /$
      $/ ENV(PROCBODY := INVDELTA(ENVA(PROCBODY), E(PROCBODY)) /$
      $/ ENV1(PROCBODY) := NULL;
         LL(PROCBODY) := LL(PROCHEAD) := LL(PROCDECL) + 1 /$
      $/ RULE(PROCDECL) := IF DN ¬= 0 THEN NULL ELSE LIST(INSTR)
         VIRDECL(PRODECL) := IF DN = 0 THEN NULL
                                       ELSE $([DN] := INSTR);
         D(PRODECL) := IF DN = 0 THEN 1 ELSE 0;
         TYPD(PROCHEAD) := PUTIN(TYPDS(TYPEP) :
                                              KIND := "PROCEDURE") /$
      $/ E1.  [SP(PROCHEAD)] := $(GENUS := TYPDS(TYPEP);
                    ATTR := CL(PROCDECL); NFORMALS := D(PROCHEAD);
                      SEGMENT := SN; ADDR := $(FORMAT := F21;
                       LN := LL(PRODECL);
                        DN := IF DN = 0 THEN DO(PRODECL) ELSE DN))) /$
      $/ E(PRODECL) := IF DN = 0 THEN E1 ELSE
                       BEGIN
                           %PUTIN(ENV1(PROCDECL) :
                                       [SP(PROCHEAD)] := FIRST(E1));
                           NULL
                       END;
         IF DN ¬= 0 THEN
         SUBORDINATE(QUALTB(PROCDECL),
                       ENV1(PROCDECL).[SP(PROCHEAD)].GENUS,
                         TYPDS(TYPEP)) /$
      $/ DO(PROCBODY) := D(PROCHEAD) + 2 /$


$P221A TYPEP ::= TYPEN


$P221B TYPEP::=
```

```
      $/ TYPDS(TYPEP).TYPE := "U" /$

$P223 PROCHEAD ::= PROCID FPPART ; MOPART SPPART

      $/ LEGIT(SPPART) := 0; CL(SPPART) := 0; DO(SPPART) := 1;
         PLACE(SPPART) := "SPECIFICATION";  DO(FPPART) := 2;
         NAMETB(MOPART) := NAMETB(SPPART) := NTB(FPPART);
         ENV1(SPPART) := ENV(PROCHEAD);
         CHECKSPEC(MATRIX(SPPART), D(FPPART);
         MAT(MOPART) := MATRIX(SPPART) /$
      $/ RULE(PROCHEAD) := CHERULES(MATRIX(MOPART), FALSE,
                                    D(FPPART), 0) /$

$P224 PROCID ::= SIGMA

$P225 FPPART ::=

      $/ D(FPPART) := 0;  NTB(FPPART) := NULL /$

$P226 FPPART ::= ( FPLIST )

$P227 FPLIST ::= FP

      $/ D(FPLIST) := 1; NTB(FPLIST).[SP(FP)] := DO(FPLIST) /$

$P228 FPLIST ::= FPLIST , FP

      $/ D(FPLIST) := D(FPLIST*) + 1 /$
      $/ NTB(FPLIST) := PUTIN(NTB(FPLIST*): [SP(FP)] := DO(FPLIST) +
                                             D(FPLIST*)) /$

$P229 FP ::= SIGMA

$P230 MOPART ::= VALPART NAMEPART

      $/ MAT(VALPART) := MATRIX(NAMEPART);
         MATRIX(MOPART) := MATRIX(VALPART) /$
      $/ IF MOAMB(NAMEPART) AND MOAMB(VALPART) THEN
           DAMB(TRUE, MOPART) ELSE
         IF MOAMB(NAMEPART) THEN DAMB(TRUE, MOPART) ELSE
         IF MOAMB(VALPART) THEN DAMB(FALSE, MOPART)/ $

$P231 MOPART ::= NAMEPART VALPART

      $/ MAT(VALPART) := MATRIX(NAMEPART);
         MATRIX(MOPART) := MATRIX(VALPART) /$
      $/ IF MOAMB(NAMEPART) AND MOAMB(VALPART) THEN
                                    DAMB(FALSE, MOPART) ELSE
```

```
      IF MOAMB(NAMEPART) THEN DAMB(FALSE, MOPART) ELSE
      IF MOAMB(VALPART) THEN DAMB(TRUE, MOPART) /$

$P232 VALPART ::= VALUE IDLIST ;

      $/ MOAMB(VALPART) := FALSE; ENV1(IDLIST) := NULL;
         LL(IDLIST) := 0;  PLACE(IDLIST) := "VALUE";
         TYPD(IDLIST) := NULL, DO(IDLIST) := 0;  CL(IDLIST) := 0 /$

$P233 VALPART ::=

      $/ MOAMB(VALPART) := TRUE;  MATRIX(VALPART) := MAT(VALPART) /$

$P234 IDLIST ::= SIGMA

      $ /L(IDLIST) := 1;
         IF PLACE(IDLIST) := "VIRTUAL" THEN SN := 0 ELSE
         BEGIN
            NEXT1 := FIND(NAMETB(IDLIST), [SP(SIGMA)]);
            IF ¬NULLB(NEXT1) THEN SN := [NEXT1] ELSE
            ERROR(SP(SIGMA), " NOT A FORMAL PARAMETER")
         END /$
      $/ MATRIX(IDLIST) := IF SN =0 THEN NULL ELSE
               $(SN = IF PLACE(IDLIST) ¬= "SPECIFICATION" THEN
                        $(MODE := PLACE(IDLIST)) ELSE
                     $(SPEC := TYPD(IDLIST); MODE:=
                       IF TYPD(IDLIST).KIND = "SIMPLE" AND
                        NULLR(FIND(TYPD(IDLIST),QUAL)) THEN "VALUE"
                                                ELSE "REFERENCE")) /$
      $/ IF PLACE(IDLIST) = "VALUE" THEN
         BEGIN
            SPEC := MAT(IDLIST).[SN].SPEC;
            IF CHECKKIND(SPEC) OR SPEC.TYPE = "REF" THEN
            ERROR(SP(SIGMA), " HAS IMPROPER MODE")
         END /$
      $/ E(IDLIST) := IF PLACE(IDLIST) = "VALUE" OR
                        PLACE(IDLIST) = "NAME" THEN NULL ELSE
                  $([SP(SIGMA)] := $(GENUS := TYPD(IDLIST);
                   ATTR := CL(IDLIST);
                    ADDR := $(FORMAT := F21; LN := LL(IDLIST);
                     DN := IF SN = 0 THEN DO(IDLIST) ELSE SN))) /$

$P235 IDLIST := IDLIST , SIGMA

      $/ IF PLACE(IDLIST) = "VIRTUAL" THEN SN := 0 ELSE
         BEGIN
            NEXT1 := FIND(NAMETB(IDLIST), [SP(SIGMA)]);
            IF ¬NULLB(NEXT1) THEN SN := [NEXT1] ELSE
             ERROR(SP(SIGMA), " NOT A FORMAL PARAMETER")
         END /$
      $/ MATRIX(IDLIST) := IF SN = 0 THEN NULL ELSE
               PUTIN(MATRIX(IDLIST*):
                   [SN] := IF PLACE(IDLIST) ¬= "SPECIFICATION" THEN
                     $(MODE := PLACE(IDLIST)) ELSE
```

```
                    $(SPEC := TYPD(IDLIST);
                      MODE := IF TYPD(IDLIST).KIND = "SIMPLE" AND
                          NULLR(FIND(TYPD(IDLIST), QUAL)) THEN
                                        "VALUE" ELSE REFERENCE)) /$
      $/ IF PLACE(IDLIST) = "VALUE" THEN
         BEGIN
            SPEC := MAT(IDLIST).[SN].SPEC;
            IF CHECKKIND(SPEC) OR SPEC.TYPE = "REF" THEN
            ERROR(SP(SIGMA), " HAS IMPROPER MODE")
         END /$
      $/ E(IDLIST) := IF PLACE(IDLIST) = "VALUE" OR
                        PLACE(IDLIST) = "NAME" THEN NULL ELSE
            UNIONDOT(E(IDLIST*), E1 *= $([SP(SIGMA)] :=
                  $(GENUS := TYPD(IDLIST); ATTR := CL(IDLIST);
                    ADDR := $(FORMAT := F21; LN := LL(IDLIST);
                     DN := IF SN = 0 THEN DO(IDLIST) ELSE SN)))) /$
      $/ L(IDLIST) := L(IDLIST*) + 1 /$
      $/ DO(IDLIST*) := DO(IDLIST) + 1 /$


$P236 NAMEPART ::= NAME IDLIST ;

      $/MOAMB(NAMEPART) := FALSE;  ENV1(IDLIST) := NULL;
        PLACE(IDLIST) := "NAME"; TYPD(IDLIST) := NULL;
        DO(IDLIST) := 0; CL(IDLIST) := 0; LL(IDLIST) := 0 /$


$P237 NAMEPART ::=

      $/ MATRIX(NAMEPART) := MAT(NAMEPART);
         MOAMB(NAMEPART) := TRUE /$


$P238 SPPART .::=

      $/RULE(SPPART) := NULL; L(SPPART) := 0; E(SPPART) := NULL;
        MATRIX(SPPART) := NULL /$


$P239 SPPART ::= SPPART SPECIFIER IDLIST ;

      $/ ENV(SPECIFIER) := ENV1(SPPART);
         RULE(SPPART) := IF PLACE(SPPART) ¬= "VIRTUAL" THEN NULL ELSE
                    APEND(RULE(SPPART*),
                          BUILDVC(TYPDS(SPECIFIER).KIND, L(IDLIST)) /$
      $/ IF PLACE(SPPART) = "VIRTUAL" THEN
         BEGIN IF ¬CHECKKIND(TYPDS(SPECIFIER)) THEN
         ERROR("INVALID SPECIFIER IN A VIRTUAL DECLARATION") END /$
      $/ IF LEGIT(SPPART) = 1 THEN
         BEGIN
            IF CHECKKIND(TYPDS(SPECIFIER)) THEN
            ERROR("INVALID SPECIFIER FOR A CLASS FORMAL PARAMETER")
         END /$
      $/ TYPD(IDLIST) := TYPDS(SPECIFIER);
         MATRIX(SPPART) := MATRIX(IDLIST);
         MAT(IDLIST) := MATRIX(SPPART*);
         L(SPPART) := L(SPPART*) + L(IDLIST) /$
      $/ DO(IDLIST) := L(SPPART*) + DO(SPPART) /$
```

```
      $/ E(SPPART) := UNIONDOT(E(SPPART*), E(IDLIST)) /$

$P240 SPECIFIER ::= TYPEN

$P241 SPECIFIER ::= ARRAY

      $/ TYPDS(SPECIFIER) := $(KIND := "ARRAY";  TYPE := "REAL") /$

$P242 SPECIFIER ::= TYPEN ARRAY

      $/ TYPDS(SPECIFIER) := PUTIN(TYPDS(TYPEN): KIND := "ARRAY") /$

$P243 SPECIFIER ::= LABEL

      $/ TYPDS(SPECIFIER).KIND := "LABEL" /$

$P244 SPECIFIER ::= SWITCH

      $/ TYPDS(SPECIFIER).KIND := "SWITCH" /$

$P245 SPECIFIER ::= PROCEDURE

      $/ TYPDS(SPECIFIER) := $(KIND := "PROCEDURE"; TYPE := "U") /$

$P246 SPECIFIER ::= TYPEN PROCEDURE

      $/ TYPDS(SPECIFIER) := PUTIN(TYPDS(TYPEN) :
                                          KIND := "PROCEDURE") /$

$P247 PROCBODY ::= ST1

      $/ BEGUN(ST1) := TRUE; OUTERMOST(ST1) := FALSE; CL(ST1) := 0 /$

$P250 CLDECL ::= PRE MPART

      $/ SN := NEWINTEGER; SM := NEWINTEGER; D(CLDECL) := 1;
         PPL(MPART) := PL(PRE);
         IF PL(PRE).ADDR.LN ¬= LL(CLDECL) THEN ERROR
          ("PREFIX AND CLASS DECLARAATIONS AT DIFFERENT LEVELS") /$
      $/ LL(MPART) := LL(CLDECL) + 2;
      $/ RULE(CLDECL) := LIST(INSTR *= $(FORMAT := F32;  SN := SM;
                                         LL := LL(CLDECL) + 1)) /$
      $/ RULE := VIRMERGE(IF PL(MPART).SEGMENT = 0 THEN RULE(MPART)
                 ELSE CONCATENATE(PL(PRE).CODE, RULE(MPART)),
                                                VIRDECL(MPART)) /$
      $/ OUTPUT(LIST(INSTR *= $(FORMAT := F31;  SN := SN;  SM := SM;
                                OBJECT := PL(PRE).OBJECT)), SM);
         OUTPUT(RULE, SN) /$
      $/ E(CLDECL).SP(MPART) := PUTIN(PL(MPART) : ADDR :=
```

```
                $(FORMAT := F21;  LN := LL(CLDECL);  DN := DO(CLDECL));
                  ATTR := CL(CLDECL);  CODE := RULE;  SEGMENT := SN;
                                                      OBJECT := SM) /$

$P251 PRE ::=

      $/ PL(PRE) := $(ADDR.LN := LL(PRE); LOCALE := NULL;
                      NFORMALS := 0;  FORMALE := NULL;  NLOCALS := 0;
                      VIRTUALE := NULL; SEGMENT := 0; OBJECT := 0 /$

$P253 MPART ::= CLASS CLID FPPART; VALPART SPPART VIRTPART CLBODY

      $/ LEGIT(SPPART) := 1;  PLACE(SPPART) := "SPECIFICATION";
         ENV1(SPPART) := ENV1(VIRTPART) := ENV(MPART);
         DO(SPPART) := 0;
         RULE(MPART) := APEND(CHERULES(MATRIX(VALPART), TRUE,
                          DO(VIRTPART) -1, PPL(MPART).NLOCALS -
                                           PPL(MPART).NFORMALS),
                           APEND(UNDECL(CLBODY), APEND(RULE(CLBODY),
                              LIST(INSTR *= $(FORMAT := F35))))))) /$
      $/ E(MPART) := UNIONDOT(E(CLBODY), E(SPPART)) /$
      $/ ENV(CLBODY) := INVDELTA(ENV(MPART),
                                 E *= #(PL(MPART).LOCALE) /$
      $/ ENV1(CLBODY) := PL(MPART).VIRTUALE /$
      $/ NAMETAB(SPPART) := NAMETB(VALPART) := NTB(FPPART);
         MAT(VALPART) := MATRIX(SPPART);
         CHECKSPEC(MATRIX(SPPART), D(FPPART)) /$
      $/ DO(FPPART) := PPL(MPART).NLOCALS + 1;
         DO(VIRTPART) := DO(FPPART) + D(FPPART);
         DO(CLBODY) := DO(VIRTPART) + L(VIRTPART) /$
      $/ PL(MPART) := $(GENUS.KIND := "CLASS";
                $/ NFORMALS := D(FPPART) + PPL(MPART).NFORMALS /$;
                $/ NLOCALS := D(CLBODY) + L(VIRTPART) + D(FPPART)
                              + PPL(MPART).NLOCALS /$;
                $/ LOCALE := INVDELTA(PPL(MPART).LOCALE, INVDELTA(
                                 PL(MPART).VIRTUALE, E(MPART))) /$;
                $/ FORMALE := INVDELTA(PPL(MPART).FORMALE,
                                       E *= # E(SPPART)) /$;
                $/ VIRTUALE := UNIONDOT(E(VIRTPART),
                                       PPL(MPART).VIRTUALE) /$) /$

$P254 VIRTPART ::=

      $/ RULE(VIRTPART) := NULL;  E(VIRTPART) := NULL;
         L(VIRTPART) := 0 /$

$P255 VIRTPART ::= VIRTUAL : SPPART

      $/ PLACE(SPPART) := "VIRTUAL"; LEGIT(SPPART) := 2;
         NAMETB(SPPART) := NULL /$

$P255A CLID ::= SIGMA
```

```
$P256 CLBODY ::= ST1

      $/ BEGUN (ST1) := TRUE;  OUTERMOST(ST1) := TRUE /$


$P257 CLBODY ::= SPLITBODY


$P258 SPLITBODY ::= INITOPS INNER : FINOPS

      $/ RULE(SPLITBODY) := APEND(ARULE(INITOPS),
                 APEND(UNDECL(INITOPS), APEND(UNDECL(FINOPS),
                   CONS(INSTR *= $(MARK := "INIT");
                    APEND(RULE(INITOPS), CONS(INSTR *=
                          $(MARK := "INNER"), RULE(FINOPS)))))))/$
      $/ VIRDECL(SPLITBODY) := UNIONR(VIRDECL(INITOPS),
                                      VIRDECL(FINOPS)) /$
      $/ E(SPLITBODY) := UNIONDOT(E(INITOPS), E(FINOPS)) /$
      $/ D(SPLITBODY) := D(INITOPS) + D(FINOPS) /$
      $/ DO(FINOPS) := DO(SPLITBODY) + D(INITOPS) /$
      $/ UNDECL(SPLITBODY) := NULL /$


$P259 INITOPS ::= BEGIN

      $/START(INITOPS) := TRUE; D(INITOPS) := 0; E(INITOPS) := NULL;
        RULE(INITOPS) := NULL;  ARULE(INITOPS) := NULL;
        UNDECL(INITOPS) := NULL; VIRDECL(INITOPS) := NULL;
        EMDEC(INITOPS) := FALSE /$


$P260 INITOPS := BLOCKHEAD ;

      $/ START(INITOPS) := TRUE;  UNDECL(INITOPS) := NULL;
         RULE(INITOPS) := NULL;  ARULE(INITOPS) := RULE(BLOCKHEAD) /$
      $/ IF EMDEC(BLOCKHEAD) THEN DAMB(FALSE, 1) /$


$P261 INITOPS ::= INITOPS ST ;

      $/ RULE(INITOPS) := APEND(RULE(INITOPS*), RULE(ST)) /$
      $/ UNDECL(INITOPS) := APEND(UNDECL(INITOPS*), RULE(ST)) /$
      $/ VIRDECL(INITOPS) := UNIONR(VIRDECL(INITOPS*), VIRDECL(ST))/$
      $/ E(INITOPS) := UNIONDOT(E(INITOPS*), E(ST)) /$
      $/ D(INITOPS) := D(INITOPS*) + D(ST) /$
      $/ DO(ST) := DO(INITOPS) + D(INITOPS*) /$
      $/ START(INITOPS) := START(INITOPS*) AND ¬EMDEC(INITOPS*) AND
                                                 FIRSTST(ST);
         IF START(INITOPS) THEN DAMB(TRUE, 1) /$


$P262 FINOPS ::= END

      $/ RULE(FINOPS) := NULL; E(FINOPS) := NULL; D(FINOPS) := 0;
         VIRDECL(FINOPS) := NULL;  UNDECL(FINOPS) := NULL /$
```

```
$P263 FINOPS ::= ; COMPT

$P264 COMPT ::= ST END
      $/ FIRSTST(COMPT) := FALSE /$

$P265 COMPT ::= ST ; COMPT

      $/ FIRSTST(COMPT) := FIRSTST(ST);
         UNDECL(COMPT) := APEND(UNDECL(ST), UNDECL(COMPT*)) /$
      $/ VIRDECL(COMPT) := UNIONR(VIRDECL(ST), VIRDECL(COMPT*)) /$
      $/ RULE(COMPT) := APEND(RULE(ST), RULE(COMPT*)) /$
      $/ E(COMPT) := UNIONDOT(E(ST), E(COMPT*))
      $/ D(COMPT) := D(ST) + D(COMPT*) /$
      $/ DO(COMPT*) := DO(COMPT) + D(ST) /$

$P265A ST ::= ST1

      $/ BEGIN(ST1) := FALSE;  ENVA(ST1) := NULL;
         OUTERMOST(ST1) := FALSE /$

$P266 ST1 ::= UNCONDST

      $/ OPEN(ST1) := "NONE" /$

$P267 ST1 ::= CONDST

      $/ FIRSTST(ST1) := FALSE;
         RULE(ST1) := PUTII(RULE(CONDST), OUTERMOST(ST1)) /$

$P268 ST1 ::= CONNST

      $/ FIRSTST(ST1) := FALSE;
         RULE(ST1) := PUTII(RULE(CONNST), OUTERMOST(ST1) /$

$P269 ST1 ::= WHILEST

      $/ FIRSTST(ST1) := FALSE
         RULE(ST1) := PUTII(RULE(WHILEST).OUTERMOST(ST1)) /$

$P270 UNCONDST ::= BASICST

      $/ RULE(UNCONDST) := PUTII(RULE(BASICST),
                                 OUTERMOST(UNCONDST))/$

$P271 UNCONDST ::= COMPST

      $/ RULE(UNCONDST) := PUTII(RULE(COMPST), OUTERMOST(UNCONDST))/$
      $/ FIRSTST(UNCONDST) := FALSE /$
```

```
$P272 UNCONDST ::= BLOCK

      $/ FIRSTST(UNCONDST) := FALSE /$


$P273 BASICST ::= UNLBASICST

      $/ VIRDECL(BASICST) := NULL;  UNDECL(BASICST);
         E(BASICST) := NULL;  D(BASICST) := 0 /$


$P274 BASICST ::= LABEL0 : BASICST

      $/ FIRSTST(BASICST) := FALSE;  SN := NEWINTEGER;
         INSTR := $(FORMAT := F21; JLABEL := SN);
         DN := CHECKVIRT(ENV1(BASICST), SP(LABEL0), "LABEL") /$
      $/ RULE(BASICST) := CONS(INSTR *= $(LABEL1 := SN),
                                RULE(BASICST*)) /$
      $/ UNDECL(BASICST) := IF DN ¬= 0 THEN UNDECL(BASICST*) ELSE
                            CONS(INSTR, UNDECL(BASICST*)) /$
      $/ VIRDECL(BASICST) := IF DN = 0 THEN VIRDECL(BASICST*) ELSE
                           PUTIN(VIRDECL(BASICST*) : [DN] := INSTR)/$
      $/ E(BASICST) := IF DN ¬=0 THEN E(BASICST*) ELSE
                 UNIONDOT(E(BASICST*), E1 *= ([SP(LABEL0)] :=
                  $(GENUS := $(KIND := "LABEL"; TYPE := "LABEL");
                     ADDR := $(FORMAT := F21;  LN := LL(BASICST);
                                          DN := D0(BASICST))))) /$
      $/ D(BASICST) := IF DN ¬= 0 THEN D(BASICST*)
                                  ELSE D(BASICST*) + 1 /$
      $/ D0(BASICST*) := IF DN ¬= 0 THEN D0(BASICST*)
                                   ELSE D0(BASICST*) + 1 /$
      $/ FIRSTST(BASICST) := FALSE / $


$P275 UNLBASICST ::= ASSST

      $/ FIRSTST(UNLBASICST) := FALSE /$


$P276 UNLBASICST ::= GOTOST

      $/ FIRSTST(UNLBASICST) := FALSE /$


$P277 UNLBASICST ::= DUMMYST

      $/ FIRSTST(UNLBASICST) := TRUE /$


$P278 UNLBASICST ::= PROCST

      $/ FIRSTST(UNLBASICST) := FALSE /$


$P280 UNLBASICST ::= OBJGEN
```

```
      $/ USE(OBJGEN) := "VALUE";   FIRSTST(UNLBASICST) := FALSE;
         RULE(UNLBASICST) := APEND(RULE(OBJGEN),
                                   LIST(INSTR *= $(FORMAT := F25))) /$

$P281 UNLBASICST ::= RESUME ( OBJEXPR )

      $/ USE(OBJEXPR) := "ADDR";   APA(OBJEXPR) := 0;
         FIRSTST(UNLBASICT) := FALSE;
         RULE(UNLBASICST) := APEND(RULE(OBJEXPR),
                                   LIST(INSTR *= $(FORMAT := F26))) /$

$P282 UNLBASICST := DETACH

      $/ RULE(UNLBASICST) := LIST(INSTR *= $(FORMAT := F27);
         FIRSTST(UNLBASICST) := FALSE;
         IF CL(UNLBASICST) = 0 THEN
         ERROR("DETACH STATEMENT IN A NON OBJECT BLOCK") /$

$P283 ASSST ::= VALASS

      $/ ALSO(VALASS) := FALSE /$

$P284 ASSST ::= REFASS

     $/ ALSO(REFASS) := FALSE /$

$P285 VALASS ::= VALLPART := VALRPART

      $/ USE(VALLPART) := "ADDR";   USE(VALRPART) := "VALUE";
         ALSO (VALRPART) := TRUE;   PL(VALASS) := PL(VALLPART);
         RULE(VALASS) := APEND(RULE(VALLPART), APEND(RULE(VALRPART),
                         LIST(INSTR *= $(FORMAT := F28;
                                         ALSO := ALSO(VALASS))))) /$
      $/ TYPE1 := PL(VALLPART).GENUS.TYPE;
         TYPE2 := PL(VALRPART).GENUS.TYPE;
         IF ¬(((TYPE1 = "INTEGER" OR TYPE 1 = "REAL") AND
               (TYPE 2 = "INTEGER" OR TYPE2 = "REAL")) OR
              (TYPE 1 = "BOOLEAN" AND TYPE2 = "BOOLEAN")) THEN
         ERROR("TYPE INCOMPATIBILITY IN A VALUE ASSIGNMENT") /$

$P286 VALLPART ::= VAR

      $/ APA(VAR) := 0;   KIND := PL(VAR).GENUS.KIND;
         IF KIND -= "ARRAY" THEN DAMB(KIND = "SIMPLE", 1) /$

$P287 VALLPART ::= PROCID2

      $/ DAMB(PL(PROCID2).GENUS.KIND = "PROCEDURE", 1);
         RULE(VALLPART) := LIST(INSTR *= $(FORMAT:= F8;
                                   ADDR := $(FORMAT := F21;  DN := 1;
                                      LN := PL(PROCID2).ADDR.LN + 1))) /$
```

```
$P287A PROCID2 ::= SIGMA

       $/ NEXT1 := FIND(ENV(PROCID2), [SP(SIGMA)]);
          IF ¬NULLB(NEXT1) THEN PL(PROCID2) := [NEXT1] ELSE
          ERROR (SP(SIGMA), " UNDECLARED IDENTIFIER") /$


$P288 VALRPART ::= VALEXPR

      $/ APA(VALEXPR) := 1/$


$P289 VALRPART ::= VALASS


$P290 REFASS ::= REFLPART := REFRPART

      $/ USE(REFLPART) := "ADDR"; USE(REFRPART) := "VALUE";
         PL(REFASS) := PL(REFLPART); ALSO(REFRPART) := TRUE;
         RULE(REFASS) := APEND(RULE(REFLPART), APEND(RULE(REFRPART),
                            LIST(INSTR := $(FORMAT := F28;
                                            ALSO := ALSO(REFRPART)S))))) /$
      $/ GENUS1 := PL(REFLPART).GENUS; GENUS2 := PL(REFRPART). GENUS;
         IF ¬(GENUS1.TYPE = "REF" AND GENUS 2.TYPE = "REF") THEN
         ERROR("TYPE INCOMPABILITY IN A REFERENCE ASSIGNMENT") ELSE
         IF GENUS1.QUAL = -1 THEN
         ERROR("LHS OF A REFERENCE ASSIGNMENT IS NONE") ELSE
         IF GENUS2.QUAL ¬= -1 THEN
         COMMENT IF THE TWO QUALS HAVE NO COMMON ANCESTOR THE
                 FUNCTION CONDQUAL REGISTERS THE ERROR;

         %CONDQUAL(QUALTB(REFASS), GENUS1.QUAL, GENUS2.QUAL) /$


$P291 REFLPART ::= VAR

      $/ APA(VAR) := 0;   KIND := PL(VAR).GENUS.KIND;
         IF KIND ¬= "ARRAY" THEN DAMB(KIND = "SIMPLE", 1) /$


$P292 REFLPART ::= PROCID2

      $/ DAMB(PL(PROCID2).GENUS.KIND = "PROCEDURE", 1);
         RULE(REFLPART) := LIST(INSTR := $(FORMAT := F8;
                                ADDR := $(FORMAT := F21; DN := 1;
                                  LN := PL(PROCID2).ADDR.LN + 1))) /$


$P293 REFRPART ::= REFEXPR

      $/ APA(REFEXPR) := 0 /$


$P294 REFRPART ::= REFASS
```

```
$P295 GOTOST  ::= GO TO DESIGEXPR

      $/APA(DESIGEXPR) := 0;
        RULE(GOTOST) := APEND(RULE(DESIGEXPR),
                              LIST(INSTR := $(FORMAT := F29))) /$


$P296 DUMMYST ::=

      $/ RULE(DUMMYST) := NULL /$


$P297 PROCST ::= PROCID1 APPART

      $/ APA(PROCID1) := 0; USE(PROCID1) := "VALUE";
         TYPE := PL(PROCID1).GENUS.TYPE;
         IF TYPE="U" THEN TYPE := "INTEGER";
         RULE(PROCST) := CONS(INSTR *= $(FORMAT := F11),
                         CONS(INSTR *= $(FORMAT := F30),
                          CONS(INSTR *= $(FORMAT := F9; OPER := TYPE),
                           APEND(RULE(APPART), APEND(RULE(PROCID1),
                             CONS(INSTR := $(FORMAT := F10),
                               LIST(INSTR *= $(FORMAT := F25)))))))) /$


$P298 CONDST ::= IFST

      $/ SN := NEWINTEGER;  OPEN(CONDST) := "NONE";
         FJUMP (IFST) := SN;
         RULE(CONDST) := APEND(RULE(IFST),
                               LIST(INSTR *= $(LABELI := SN))) /$


$P299 CONDST ::= IFST ELSE ST

      $/ SN := NEWINTEGER;  SM := NEWINTEGER;  FJUMP(IFST) := SN;
         OPEN(CONDST) := OPEN(ST);
         RULE(CONDST) := APEND(RULE(IFST),
                               FIXCOND(RULE(ST), SM, SN)) /$
      $/ D(CONDST) := D(IFST) + D(ST) /$
      $/ D0(ST) := D(IFST) + D0(CONDST) /$
      $/ UNDECL(CONDST) := APEND(UNDECL(IFST), UNDECL(ST)) /$
      $/ VIRDECL(CONDST) := UNIONR(VIRDECL(IFST), VIRDECL(ST)) /$
      $/ E(CONDST) := UNIONDOT(E(IFST), E(ST)) /$


$P300 CONDST ::= IFCL CONNST

      $/ SN := NEWINTEGER; FJUMP(IFCL) := SN;
         RULE(CONDST) := APEND(RULE(IFCL), APEND(RULE(CONNST),
                               LIST(INSTR *= $(LABELI := SN)))) /$


$P301 CONDST ::= IFCL WHILEST

      $/ SN := NEWINTEGER; FJUMP(IFCL) := SN;
         RULE(CONDST) := APEND(RULE(IFCL), APEND(RULE(WHILEST),
                               LIST(INSTR *= $(LABELI := SN))) /$
```

```
$P302 CONDST ::= LABEL0 : CONDST

      $/ SN := NEWINTEGER;   INSTR := $(FORMAT := F20;   JLABEL := SN);
         DN := CHECKVIRT(ENV1(CONDST), SP(LABEL0), "LABEL") /$
      $/ RULE(CONDST) := CONS(INSTR *= $(LABELI := SN),
                                          RULE(CONDST*)) /$
      $/ D(CONDST) := IF DN ¬= 0 THEN D(CONDST*)
                                 ELSE D(CONDST*) + 1 /$
      $/ DO(CONDST*) := IF DN ¬= 0 THEN DO(CONDST)
                                   ELSE DO(CONDST) + 1 /$
      $/ UNDECL(CONDST) := IF DN ¬= 0 THEN UNDECL(CONDST*) ELSE
                                    CONS(INSTR, UNDECL(CONDST*)) / $
      $/ VIRDECL(CONDST) := IF DN = 0 THEN VIRDECL(CONDST*) ELSE
                           PUTIN(VIRDECL(CONDST*) : [DN] := INSTR) /$
      $/ E(CONDST) := IF DN ¬= 0 THEN E(CONDST*) ELSE
              UNIONDOT(E(CONDST*), E1 *= $([SP(LABEL0)] :=
                 $(GENUS := $(KIND:= "LABEL"; TYPE := "LABEL");
                   ADDR := $(FORMAT := F21; LN := LL(CONDST);
                                          DN := DO(CONDST))))) /$


$P303 IFST ::= IFCL UNCONDST

      $/ ENVA(UNCONDST) := NULL; BEGUN(UNCONDST) := FALSE;
         CL(UNCONDST) := 0;   OUTERMOST(UNCONDST) := FALSE;
         RULE(IFST) := APEND(RULE(IFCL), RULE(UNCONDST)) /$


$P304 IFCL ::= IF BEXPR THEN

      $/ USE(BEXPR) := "VALUE"; APA(BEXPR) := 0;
         RULE(IFCL) := APEND(RULE(BEXPR), LIST(INSTR *= $(FORMAT :=
                         F17; JLABEL := FJUMP(IFCL)))) /$


$P305 WHILEST ::= WHILE BEXPR DO ST

      $/ APA(BEXPR) := 0;   USE(BEXPR) := "VALUE";
         SM := NEWINTEGER;   SN := NEWINTEGER
         RULE(WHILEST) := CONS(INSTR *= $(LABELI := SN),
                     APEND(RULE(BEXPR),
                      CONS(INSTR *= $(FORMAT := F17; JLABEL := SM),
                       APEND(RULE(ST),
                        CONS(INSTR *= $(FORMAT := F1;   JLABEL := SN),
                         LIST(INSTR *= $(LABELI := SM)))))))) /$


$P306 WHILEST ::= LABEL0 : WHILEST

      $/ SN := NEWINTEGER; INSTR := $(FORMAT := F20; JLABEL := SN);
         DN := CHECKVIRT(ENV1(WHILEST), SP(LABEL0), "LABEL") /$
      $/ RULE(WHILEST) := CONS(INSTR *= $(LABELI := SN),
                                          RULE(WHILEST*)) /$
      $/ D(WHILEST) := IF DN ¬= 0 THEN D(WHILEST*)
                                   ELSE D(WHILEST*) + 1 /$
      $/ DO(WHILEST*) IF DN ¬= 0 THEN DO(WHILEST)
```

```
                                ELSE DO(WHILEST) + 1 /$
      $/ UNDECL(WHILEST) := IF DN ¬= 0 THEN UNDECL(WHILEST*) ELSE
                                    CONS(INSTR, UNDECL(WHILEST*)) /$
      $/ VIRDECL(WHILEST) := IF DN = 0 THEN VIRDECL(WHILEST*) ELSE
                           PUTIN(VIRDECL(WHILEST*) : [DN] := INSTR) /$
      $/ E(WHILEST) := IF DN ¬= 0 THEN E(WHILEST*) ELSE
               UNIONDOT(E(WHILEST*), E1 *= $([SP(LABELO)] :=
                  $(GENUS := $(KIND := "LABEL"; TYPE := "LABEL");
                     ADDR := $(FORMAT := F21; LN := LL(WHILEST);
                                          DN := DO(WHILEST))))) /$


$P307 CONNST ::= INSPECT OBJEXPR CONNPART OTCL

      $/ SM := NEWINTEGER;  SN := NEWINTEGER;
         D(CONNST) := D(OTCL) + 1 /$
      $/ UNDECL(CONNST) := CONS(INSTR *= $(FORMAT := F9;
                                    OPER := "REF"), UNDECL( OTCL)) /$
      $/ O(OTCL) := OPEN(CONNPART); OPEN (CONNST) := OPEN(OTCL) /$
      $/ USE(OBJEXPR) := "VALUE"; FJUMP(CONNPART) := SN;
         TJUMP(CONNPART) := SM;  APA(OBJEXPR) := 0;
         DO(OTCL) := DO(CONNST) + 1 /$
      $/ RULE(CONNST) := CONS(INSTR *= $(FORMAT := F8;
            ADDR := $(FORMAT := F21;  LN := LL(CONST);
                     DN := DO(CONNST))), APEND(RULE(OBJEXPR),
              CONS(INSTR *= $(FORMAT := F28; ALSO := FALSE),
               APEND(RULE(CONNPART), CONS(INSTR *= $(LABELI := SN),
                APEND(RULE(OTCL),
                  LIST(INSTR *= $(LABELI := SM)))))))) /$


$P308 CONNST ::= INSPECT OBJEXPR DO CONNBLOCK2 OTCL

      $/ SN := NEWINTEGER;  SM := NEWINTEGER;  SL := INTEGER
         D(CONNST) := D(OTCL) + 1 /$
      $/ UNDECL(CONNST) := CONS(INSTR *= $(FORMAT := F9;
                                    OPER := "REF"), UNDECL(OTCL)) /$
      $/ OUTPUT(APEND(UNDECL(CONNBLOCK2), APEND(RULE(CONNBLOCK2),
                           LIST(INSTR *= $(FORMAT := F12)))), SL) /$
      $/ O(OTCL) := OPEN(CONNBLOCK2);  OPEN(CONNST) := OPEN(OTCL) /$
      $/ USE(OBJEXPR) := "VALUE"; APA(OBJEXPR) := 0;
         BEGUN(CONNBLOCK2) := TRUE; DO(CONNBLOCK2) := 1;
         DO(OTCL) := DO(CONNST) + 1 /$
      $/ LL(CONNBLOCK2) := LL(CONNST) + 1 /$
      $/ E(CONNST) := E(OTCL) /$
      $/ ENV(CONNBLOCK2) := INVDELTA(ENV(CONNST),
            INVDELTA([QUALTB(CONNST).[PL(OBJEXPR). GENUS.QUAL].
                           CLASSN].LOCALE), E(CONNBLOCK2))) /$
      $/ ADDR := $(FORMAT := F21;  LN := LL(CONNST);
                                    DN := DO(CONNST)) /$
      $/ ITEM (CONNBLOCK2) := CONS(XX *= $(QUAL :=
            PL(OBJEXPR).GENUS.QUAL;  ADDR := ADDR), ITEM((CONNST)) /$
      $/ RULE(CONNST) := CONS(INSTR *= $(FORMAT := F2; ADDR := ADDR),
           APEND(RULE(OBJEXPR),
            CONS(INSTR *= $(FORMAT := F28; ALSO := TRUE),
             CONS(INSTR *= $(FORMAT := F16; OPER := "=/="),
              CONS(INSTR *= $(FORMAT := F17; JLABEL := SN),
               CONS(INSTR *= $(FORMAT := F11),
```

```
                CONS(INSTR *= $(FORMAT := F22;
                                LN := LL(CONNBLOCK2);  SN := SL),
                 CONS(INSTR *= $(FORMAT := F1;  JLABEL := SM),
                  CONS(INSTR *= $(LABEL := SN), APEND(RULE(OTCL),
                   LIST(INSTR *= $(LABELI := SM)))))))))))) /$

$P309 CONNST ::= LABEL0 : CONNST

      $/ SN := NEWINTEGER;  INSTR := $(FORMAT := F20;  JLABEL := SN);
         RULE(CONNST) := CONS(INSTR *= $(LABELI := SN),
                                        RULE(CONNST*)) /$
      $/ DN := CHECKVIRT(ENV1(CONNST), SP(LABEL0), "LABEL") /$
      $/ UNDECL(CONNST) := IF DN ¬= 0 THEN UNDECL(CONNST*) ELSE
                              CONS(INSTR, UNDECL(CONNST*)) /$
      $/ VIRDECL(CONNST) := IF DN ¬= 0 THEN VIRDECL(CONNST*) ELSE
                            PUTIN(VIRDECL(CONNST) : [DN] := INSTR) /$
      $/ D(CONNST) := IF DN ¬= 0 THEN D(CONNST*)
                                 ELSE D(CONNST*) + 1 /$
      $/ DO(CONNST*) := IF DN ¬= 0 THEN DO(CONNST)
                                   ELSE DO(CONNST) + 1 /$
      $/ E(CONNST) := IF DN ¬= 0 THEN E(CONNST*) ELSE
           UNIONDOT(E(CONNST*), E1 *= $([SP(LABEL0)] :=
                 $(GENUS := $(KIND := "LABEL"; TYPE := "LABEL");
                   ADDR := $(FORMAT := F21; LN := LL(CONNST);
                                            DN := DO(CONNST)))) /$

$P310 CONNPART := CONNCL

$P311 CONNPART ::= CONNPART CONNCL

      $/ SN := NEWINTEGER;  FJUMP(CONNPART*) := SN;
         RULE(CONNPART) := APEND(RULE(CONNPART*), CONS(INSTR *=
                                 $(LABEL1 := SN), RULE(CONNCL)) /$
      $/ OPEN(CONNPART) := OPEN(CONNCL) /$

$P312 CONNCL ::= WHEN CLID1 DO CONNBLOCK1

      $/ SN := NEWINTEGER;  BEGUN(CONNBLOCK1) := TRUE;
         DO(CONNBLOCK1) := 1;
         ADDR := $(FORMAT := F21;  LN := LL(CONNCL);
                                   DN := DO(CONNCL));
         ITEM(CONNBLOCK1) := CONS(XX *= $(QUAL := PL(CLID1).SEGMENT;
                                  ADDR := ADDR), ITEM(CONNCL)) /$
      $/ RULE(CONNCL) := CONS(INSTR *= $(FORMAT := F8; ADDR := ADDR),
           CONS(INSTR *= $(FORMAT := F8; ADDR := PL(CLID1).ADDR),
            CONS(INSTR *= $(FORMAT := F16;  OPER := "IN-WHEN"),
             CONS(INSTR *=(FORMAT := F17; JLABEL := FJUMP(CONNCL)),
              CONS(INSTR *= $(FORMAT := F11);
               CONS(INSTR *= $(FORMAT := F22; LL := LL(CONNBLOCK1);
                                              SN := SN),
                LIST(INSTR *= $(FORMAT := F1;
                                JLABEL := TJUMP(CONNCL))))))))) /$
    $/ OUTPUT(APEND(UNDECL(CONNBLOCK1), APEND(RULE(CONNBLOCK1),
                           LIST(INSTR *= $(FORMAT := F12)))), SN) /$
```

```
      $/ LL(CONNBLOCK1) := LL(CONNCL) + 1 /$
      $/ ENV(CONNBLOCK1) := INVDELTA(ENV(CONNCL),
                            INVDELTA(PL(CLID1).LOCALE, E(CONNBLOCK1))) /$

$P313 CONNBLOCK1 ::= ST1

      $/ ENV1(ST1) := NULL;   ENVA(ST1) := ENV(CONNBLOCK1);
         OUTERMOST(ST1) := FALSE;   CL(ST1) := 0 /$

$P314 CONNBLOCK2 := ST1

      $/ ENV1(ST1) := NULL;   ENVA(ST1) := ENV(CONNBLOCK2);
                              OUTERMOST(ST1) := FALSE;   CL(ST1) := 0; /$

$P315 OTCL ::=

      $/ RULE(OTCL) := NULL;   UNDECL(OTCL) := NULL;
         VIRDECL(OTCL) := NULL; D(OTCL) := 0;   E(OTCL) := NULL;
         OPEN(OTCL) := IF O(OTCL) = "CLOSED" OR
                          O(OTCL) = "OPENDISAMB" THEN "OPENDISAMB"
                                                 ELSE "OPEN";
         IF OPEN(OTCL) = "OPENDISAMB" THEN DAMB(TRUE, 1) /$

$P316 OTCL ::= OTHERWISE ST

      $/ IF O(OTCL) = "OPEN" OR O(OTCL) = "OPENDISAMB" THEN
         DAMB(FALSE, 1) /$
      $/ OPEN(OTCL) := IF OPEN(ST) = "OPENDISAMB" THEN "OPEN" ELSE
                       IF OPEN(ST) = "NONE" THEN "CLOSED"
                                            ELSE OPEN(ST) /$
```

4.2 ANALYSIS OF THE DEFINITION

This section analyses the differences between Wilner's and the present definition. Only major differences are analysed in detail; differences arising from minor errors or omissions are noted but not commented. It should be noted that the present definition implements only a subset of Wilner's definition. The productions for real

numbers, characters and simulation were left out. The reason for this omission is not any limitation imposed by FOLDS; it simply reflects an individual desire to restrict as much as possible the amount of work to be done. It also should be noted that the input-output rules are missing; they were not included because they would involve the hand coding of a large set of target language instructions which did not seem to be very relevant to the purposes of the present definition.

4.2.1 AMBIGUITIES

In Wilner's definition, ambiguities are handled in essentially the same fashion as errors. The definition had thus to be changed to adapt it to SPINDLE's formalism for handling ambiguities; a number of new attributes were introduced (APA, DAR, EMDEC, AEMDEC, NUMDEC, START, SID), one was eliminated (OUTER), and another modified (FIRSTST, from inherited to synthesized). Furthermore, some ambiguities were detected that had not been noted by Wilner: one arising from an empty name part and/or value part in a procedure heading; one arising from the first statement in the compound tail of a block being empty; and one arising from the first statement in INITIAL OPERATIONS being empty.

4.2.2 QUALTB

The attribute QUALTB maps the segment designation of a class into the symbol table entry for the class and into the class's prefix class. It is introduced to simplfy the implementation of a series of functions specified by Wilner (e.g. CONDQUAL). As a consequence of its introduction, Wilner's functions CPL and IDSP are not implemented since the values they would return can be directly obtained from QUALTB.

4.2.3 VIRTUALS

A number of modifications were introduced due to errors found in Wilner's scheme for handling virtual (SIMULA) attributes. The attribute ENV1 was introduced to avoid the following circularity arising in Wilner's definition: when checking if an identifier is virtual in an identifier declaration the attribute ENV is used to check if the identifier is virtual; however, ENV depends on the attribute E whose value depends on the test on the attribute ENV. In the modified scheme the test is made upon ENV1 which does not depend on E. The function VIRMERGE had to be modified since the original version does not work when the attributes of a class include an array. The object code generated from procedure statements was also modified. In Wilner's definition, different rules are generated if the procedure is a proper or a typed procedure; thus, a virtual

proper procedure which is redefined as a typed procedure, will cause execution errors for all procedure statements that call the procedure from the body of the prefix class. As modified, the procedure statement always generates the same object code. The final modification was the introduction of the function SUBORDINATE, which checks the subordination rules for the redefinition of virtual procedures; it is missing in Wilner's definition.

4.2.4 CLASS CONCATENATION

The class concatenation mechanism proposed by Wilner does not work when a class has formal parameters and is prefixed. The implementation of a valid mechanism, besides changing the definition, required some of the changes in Wilner's machine which were explained at the beginning of this chapter.

4.2.5 FUNCTION INVDELTA

This function is a modified version of Wilner's ↑ function. While not wrong, Wilner's function was more complicated than necessary. INVDELTA simply implements the ALGOL rules for renaming global variables inside a block.

4.2.6 CODE

As proposed by Wilner the CODE function does not work. Instead of implementing a function, an attribute CODE is included in the symbol table entry of a class: its value is the rule generated for the class.

4.2.7 ARRAY DECLARATIONS

According to the SIMULA definition [DMN 70], the array bounds in an array declaration may contain variables (or procedures) that are global to the block to which the array belongs, plus formal parameters, if the array is declared in a class or procedure body. The attribute ENVA was introduced to implement this feature, which is ignored by Wilner.

As defined by Wilner, an array segment having more than one array identifier will not generate the proper code. The correction of this error necessitated the changes in the machine instruction MAK which were explained at the beginning of the current chapter.

4.2.8 LABELLED BLOCKS

Production P175 is significantly different from the corresponding production in Wilner's definition which had a substantial number of errors. The attribute NOLABEL, had to be introduced to detect the leftmost label when more than one label appeared on a block.

Production 177 and 178 were dropped and 176 replaced by

BLOCK ::= UNLABELLED PREFIXED BLOCK

a labelled prefixed block causes an ambiguity and nothing is lost, semantically, by changing the grammar.

4.2.9 PROCEDURE AND CLASS HEADINGS

The mechanisms proposed by Wilner for headings (using the attributes MAT, MATRIX and VECT and the function m), while not wrong, would have been cumbersome to implement in SPINDLE. A similar but simpler mechanism is implemented using the attributes MAT, MATRIX, NAMETB and NTB, and no special functions.

4.2.10 PROCEDURE DECLARATIONS

To simplify the definition, productions

221: PROCDECL ::= PROCEDURE PROCHEAD PROCBODY

222: PROCDECL ::= TYPEN PROCEDURE PROCHEAD PROCBODY

were replaced by productions

P221: PROCDECL ::= TYPEP PROCEDURE PROCHEAD PROCBODY

P221A: TYPEP ::= TYPEN

P221B: TYPEP ::=

This modification does not alter the content of the definition but serves to point out how a proper choice of the grammar can result in a more compact SPINDLE definition.

4.2.11 ST1

The introduction of the nonterminal ST1 is another modification done for the purpose of having a more compact definition. The use of both ST1 and ST decreases the number of semantic rules necessary in the definition. Thus a number of attributes of ST1 have values assigned to them in P265A; if only ST were used, those values would have to be assigned in every production in which ST were a RHN.

4.2.12 OTHER MODIFICATIONS

Besides the productions noted above, the following productions had to be modified due to errors or omissions in Wilner's definition: P3, P49, P83, P89, P90, P159, P160, p169, P170, P179, P180, P181, P183, P210, P211, P218, P219, P221, P223, P232, P234, P235, P236, P239, P245, P250, P251, P253, P256, P258, P259, P261, P262, P265, P267, P268, P269, P270, P271, P285, P290, P299, P303, P307, P308, P312, P314, P315, P316.

CHAPTER 5

CONCLUSION

The preceding chapters presented a description of FOLDS and of its applications. The current chapter reviews the system, describing its present implementation status and pointing out needed improvements; it also indicates some areas for further research.

The system implements and extends Knuth's method for the formal definition of semantics, incorporating Wilner's extensions to the method. The declarative nature of the method is preserved by the use of a special control structure wich permits a nearly complete dissociation between language definition and compilation. A formalism for the semantic resolution of syntactic ambiguities is introduced together with appropriate control mechanisms to carry out the disambiguation processes. The actual disambiguation mechanism is transparent in the definition as is the compilation carried out from it. The system provides a language, SPINDLE, for writing the definitions and a machine, MUTILATE, to compile strings of the language directly from the definition. The language incorporates a flexible data structure representation; a syntax specification mechanism imposing practically no restrictions on the user; a set of semantic primitives necessary for specifying the semantic rules associated with each production. The language provides the necessary composition rules so that new semantic operators can be built from the primitives provided by the system.

As shown by the definition of SIMULA in Chapter 4, MUTILATE is capable of handling the definition of large languages and the compilation of sizeable programs in the defined languages. Further on, a series of improvements are suggested to increase the capacity of the system. However, in its present stage, the size of the programs it is capable of compiling is quite adequate for the primary purpose of the machine, which is to check the correction of definitions. A series of debugging aids are incorporated in the mechanism and have proved adequate in the debugging of the SIMULA definition: however this is a biased opinion since the debugging of the system was carried out in parallel with the debugging of the definition and no other user besides the author has used the system.

The system is currently implemented on an IBM 360/67 and occupies 280K bytes of sterage. It consists of the MUTILATE assembler and the MUTILATE machine, both written in PL/360 [Ma 71]. The SPINDLE compiler has not yet been implemented; SPINDLE programs are hand-compiled into MUTILATE assembler code. The assembler incorporates most of the important features of the compiler (e.g., the generation of implied semantic rules), so the hand compilation is very straightfoward. The SIMULA definition compiles into approximately 8000 assembler instructions which take 0.13 minutes of CPU time to assemble. The machine is implemented as two separate programs: the first implements the parser and the lexical analyzer while the second implements the MUTILATE interpreter. The SIMULA definition occupies approximately 30K of byte addressed memory, out of a maximum of 64K which indicates that there are no practical limitations on the size of languages that can be defined and run in FOLDS. One real limitation is the size of the programs of the defined language that

can be compiled by MUTILATE; while compilation time does not seem to be a constraint 'see the timings that accompany the SIMULA programs in Appendix 4), space definitely is; with the present storage (280K bytes), the largest SIMULA program that can be compiled is about 30% larger than program X in Appendix 4. However, this size of program is more than adequate for the purposes of the system, which is to test the definition of languages; it is certainly not adequate for a production compiler.

The experience with the system is somewhat limited since the only practical language defined in it is SIMULA 67. Also the restrictions imposed upon the SIMULA definition (that it should follow the SIMULA 67 grammar and approximate Wilner's definition) makes it difficult to generalize from the present experiment. Inasmuch as SIMULA is representative of a large class of programming languages, the system seems perfectly adequate for their definition. However much more experience is needed before definitive conclusions can be drawn about the adequacy of the system for a broader class of languages.

Despite the disclaimers, initial experience with the system has been very encouraging. The discipline involved in writing a definition formally has paid off handsomely in avoiding and detecting dozens of errors and inconsistencies in the previous definition of SIMULA. There have been many advantages in having a working syystem since many of the errors in Wilner's definition could hardly have been noticed by hand since humans are not so demanding in precision. Although space is limited, in fac the limitation was not so severe as expected. since programs nearly a 100 lines long can be handled; this is almost an order of magnitude better than was expected. The

running speed is also quite satisfactory. FOLDS has proved its power and flexibility with the definition of SIMULA. Finally, the inovations introduced in FOLDS, such as parallel statements and the disambiguation mechanisms, seems to be working rather well.

Further research is needed to establish SPINDLE programming techniques. The SIMULA definition has what seems to be very adequate techniques for the handling of labels and symbol tables but the handling of ambiguities seems to be a bit cumbersome. A further study of the attributes used in the definition should also reveal areas for improvement such as a reduction in the number of attributes and a simplification of the user-defined functions by the utilization of more adequate attributes. An example of this type of simplification is the introduction of QUALTB on Wilner's definition of SIMULA.

Another area for research is the balance between syntax and semantics. In the present definition the syntax was mostly fixed by the decision to stick to the official SIMULA grammar. While it served to show the power of the method it complicated the definition and made it harder to understand. A joint design of the syntax and semantics would obviously yield simpler and more readable definitions. As of now the SIMULA definition seems to be somewhat hard to understand for someone not familiar with the SIMULA language; if indeed this is true, a better choice of syntax and better SPINDLE programming techniques should help. Also a more liberal use of comments would certainly improve the readability of the definition.

It should be noted that in the SIMULA definition a significant amount amount of the code is dedicated to error and ambiguity handling. Since this portion of the code is a result of the design of the syntax, it is easy to see how a better syntax design can decrease the complexity of definitions.

Another area for further research is the choice of the target language. In the SIMULA definition, Wilner chose as target language the order code of a machine similar to the one defined by Randell & Russell [RR 64] for ALGOL 60. In a sense this is unfortunate since the machine is complicated enough to make understanding it nontrivial, thus obscuring some aspects of the definition. However, as pointed out by Knuth [Kn 71], the target language should be of a high enough level so that the issues involved in the definition are not obscured by the level of detail made necessary by the low level of the target language. There are tradeoffs in the choice of the target language and further research is needed to establish criteria for a proper choice. One possible choice is to compile directly into some mathematical formalism, such as the one proposed by Scott [SS 71], which then directly gives the meaning of the strings of the defined language. An additional advantage of this choice is that proofs about the programs can then be worked out directly. The disadvantage of this choice of target language is that it is not very relevant to the compiler writer, who should be one of the main users of a language definition.

It should be noted that since the target language, from the point of view of the language designer, is essentially debugging information, it should be made as symbolic as possible. For instance, in the definition of SIMULA, it would have been very helpfull to link the source statements to the target language they generate; while this involves changing the definition, it involves only minor changes and should be possible to effect with relative ease.

As described in Chapter 3, the parsing and filling in of the semantics are performed in two separate steps; this approach was

chosen for its simplicity and because at the time the decision was reached the processes involved in the filling of the semantics were not completly understood. But a one step approach (parsing and filling in if semantics simulataneously), if successful, could both reduce the compilation time and increase the size of programs that can be compiled by the system; the amount of backtracking and the number of ambiguous subtrees generated could both be reduced. A new parsing scheme will probably have to be chosen since Earley's, as analysed in Chapter 4, does not seem to adapt itself well to a one step scheme.

Another aspect of the system that deserves further study is the DEVELOP function which traverses the parse tree, returning a different node for each call. A garbage collection mechanism collects all those nodes for which all associated parallel processes have terminated and those attributes whose values are not relevant to any other attributes. Thus, the order in which the nodes are developed is critical for efficient space management. As now implemented, DEVELOP traverses the tree in a top-down, left to right order, which reflects the bias of most programming languages. But in SIMULA, for instance, a procedure body may use a variable whose declaration comes to the right of the procedure declaration; this shows that the left to right bias is not all-pervasive. In terms of the definition in Chapter 4, the ENV attribute for the procedure body will be defined only after all the declarations at the same level have been processed. In this case it would clearly be more efficient to postpone the development of the subtree corresponding to the procedure body until ENV(PROCDECL) had been defined. As can be perceived, a "smarter" DEVELOP function can increase the size of programs that can be compiled by the system.

Another necessary improvement to the system is the introduction of the data types REAL and STRING and the necessary functions for their manipulation. While not essential, these features should increase the flexibility of the system. The system was designed with these data types in mind and so their inclusion will result in additions to the system, but not many changes.

In the present implementation all the output is performed at the end of a run; this guarantees that all attributes are defined before they are printed. This results in a great waste of space; a control structure that would output attributes as soon as they are defined, while preserving the output rules stated in 2.9.1, could greatly improve the capacity of MUTILATE.

The power of FOLDS could be greatly increased by the use of a more powerful scheme for the description of the syntax. For example, either the scheme proposed by Galler & Perlis [GP 70] or the one used by Floyd to describe the syntax of ALGOL W [Si 71] would be convenient. Such schemes, besides permitting a more compact description of the syntax of a language, generate shallower parse trees for any given string of the language, and thereby minimize the number of attributes passed from node to node. The use of a simple production scheme for the grammar necessitates the use of intermediary nonterminals which also increase the size of the tree. The use of a more powerful syntax scheme should also reduce the number of attributes by decreasing the amount of information to be circulated through the tree. However, the adoption of these more powerful syntax schemes is not trivial since a set of semantic operators will have to be created for the manipulation of attributes. Research is needed to choose the appropiate syntax scheme and to

choose and develop the associated semantic operators. The adoption of a new syntax specification method implies a large overhaul of the present system which should, however, serve as the basis for the improved system.

The checking of definitions is another area for further research. Given a language and its definition, how should a set of programs in the language be chosen to assure that the definition is well formed and that it actually reflects the language designer's concepts about the language? It must be possible, given the definition of a particular language to devise a systematic approach, so that if not all, at least nearly all possible elementary constructs of the language can be checked out. The experience acquired with SIMULA seems to indicate that FOLDS is capable of compiling programs long enough to test the definition and that the debugging aids in the system seem adequate enough for the task. But, although a large number of tests have been performed the definition probably still contains some undetected errors. The experience also shows that the tests should be performed with the programs as small as possible; in a language as large as SIMULA, even small programs generate large parse trees and a great number of attributes. It is thus very hard to keep track of all that is going on during a MUTILATE run.

Another area for further research is in the development of production compilers directly from a SPINDLE definition. While the stress in FOLDS is towards generality, definitions could be classified according to their semantic and syntactic characteristics, and efficient compilers could be generated for certain categories. Local code optimization can be easily achieved with the use of

appropriate attributes, and a special category of rules could be introduced to help generate efficient compilers. Ideally, it should be possible to generate an efficient compiler directly from the definition, without any further information; however, this does not seem realistic, at least at the moment. It should be noted that a nondeterminisc approach such as this is bound to be inherently less efficient than deterministic approaches.

BIBLIOGRAPHY

[AU 71] Aho, A.V. and Ullman, J.D., "Translations on a context free grammar.", Info. and Control 19, 5 (Dec. 1971), 439-475

[BM 62] Brooker, R.A. and Morris, D., "A general translation program for phrase structured languages.", J. ACM 9, 1 (Jan. 1962), 1-10

[DH 72] Dahl, O.-J. and Hoare, C.A.R., "Hierarchical program structures.", in Structured Programming, by O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, London, 1972, pp-175-220.

[Di 72] Dirksen, J.A., "The QA4 Primer.", SRI project 8721 DRAFT memo 15 June 1972

[DMN 70] Dahl, O.-J., Myhrhaug, B. and Nygaard, K., SIMULA 67 Common Base Language, Publication No. S-22, Norwegian Computing Center, October 1970

[Ea 68] Earley, J., "An efficient context free parsing algorithm.", Comm. ACM 13, 2(Feb. 1970), 94-102

[Fe 66] Feldman, J.A., "A formal semantics for computer languages and its application to compiler-compilers.", Comm. ACM 9, 1 (Jan. 1966), 3-9

[Fe 72] Feldman, J.A. et al, "Recent developments in SAIL, an Algol-based language for artificial intelligence.", Proc. FJCC (1972)

[Fi 70] Fisher, D.A., "Control structures for programming languages.", Ph.D. thesis, Carnegie-Mellon University, May 1970

Reproduced from best available copy.

[Fl 62] Floyd, R.W., "On the nonexistence of a phrase structure grammar for ALGOL 60.", Comm. ACM 5, 9 (Sep. 1962), 483-484

[GP 70] Galler, B.A. and Perlis, A.J., A View of Programming Languages, Addison-Wesley, Reading, Mass., 1970

[He 71] Hewitt, C., "Procedural Embedding of knowledge in Planner.", Proc. Second IJCAI, September 1971, pp. 167-182

[HU 69] Hopcroft, J, and Ullman, J., Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass., 1969

[IM 72] Ichbiah, J.D. and Morse, S.P., "General concepts of the SIMULA 67 programming language.", Annual Review in Automatic Programming, V.7, Part 1, 1972, pp. 65-93

[Ir 63] Irons, E.T., "Towards more versatile mechanical translators.", Proc. Sympos. Applied Math., 1963, V. 15, pp. 41-50

[Jo 68] Johnson, W.L. et al, "Automatic generation of efficient lexical processors using finite state techniques.", Comm. ACM 11, 12 (Dec. 1968), 805-813

[Kn 68a] Knuth, D.E., "Semantics of context free languages.", Mathematical Systems Theory J. 2, 2 (1968), 127-145

[Kn 68b] Knuth, D.E., The Art of Computer Programming, vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. 1969

[Kn 71] Knuth, D.E., "Examples of formal semantics.", in Symposium on Semantics of Algorithmic Languages, Spring-Verlag, New York, 1971, pp. 212-235

[LLS 68] Lucas, P., Lauer, P. and Stigleitner, H., "Method and notation for the formal definition of programming languages.", TR25.087, IBM Lab. Vienna, 1968

[Ma 71] Malcolm, M.A., "PL360 (Revised)- A programming language for

the IBM /360.", Technical Report No. 215, Computer Science Department, Stanford University (May 1971)

[MCa 65] McCarthy, J. et al, The LISP 1.5 Programming Manual, MIT Press, Cambridge, Mass., 1965

[MCl 65] McClure, R.M., "TMG- A syntax directed compiler.", Proc. ACM Nat. Conf., 1965, V. 20, pp. 262-274

[RR 64] Randell, B. and Russell, L.J., ALGOL 60 Implementation, Academic Press, London, 1964

[Si 71] Sites, R.L., "Algol W Reference Manual.", Technical Report No. 230, Computer Sciences Department, Stanford University (August 1971)

[SS 71] Scott, D. and Strachey, C., "Toward a mathematical semantics for computer languages.", Proc. of the Symposium on Computers and Automata, Microwave Research Institute Syposium Series Vol. 21, Polytechnic Institute of Brooklin, 1971

[We 72] Wegner, P., "The Vienna Definition Language.", ACM Computing Surveys 4, 1 (Mar. 1972), 5-63

[Wi 71] Wilner, W.T., "A declarative semantic definition.", Ph.D. thesis, Computer Science Department, Stanford University, 1971

[Wi 72] Wilner, W.T., "Formal semantic definition using synthesized and inherited attributes.", in Formal Semantics of Programming Languages, Prentice Hall, Englewood Cliffs, New Jersey, 1972, pp. 25-40

[WW 66] Wirth, N., and Weber, H., "EULER: A generalization of Algol and its formal definition part I.", Comm. ACM 9, 1 (Jan. 1966), 13-23; "Part II.", Comm. ACM 9, 2 (Feb. 1966), 89-99

APPENDIX 1

```
COMMENT THIS IS A DESCRIPTION OF TURINGOL IN SPINDLE. THE LIST
        ATTRIBUTE OBJPROG HOLDS A LIST OF CONSTRUCTS THAT ARE
        TRANSFORMED INTO A TL/I PROGRAM BY THE PROCEDURE OUTPUT.
        LABELS ARE HANDLED BY MEANS OF LABEL-VALUES. TO EACH LABEL
        IS ASSOCIATED A UNIQUE INTEGER, THE LABEL VALUE, AND A
        PSEUDO-INSTRUCTION (WITH COMPONENT TAG) IS INSERTED IN FRONT
        OF A LABELED INSTRUCTION. THE VALUE OF TAG IS THE LABEL-
        VALUE. THE OUTPUT PROCEDURE THEN BINDS LABEL-VALUES AND
        ADDRESSES BY MEANS OF THE MAP ATTRIBUTE. PSEUDO INSTRUCTIONS
        ARE NOT PRINTED. THE SYMBOL TABLE IS REPRESENTED BY THE
        CONSTRUCTS E AND ENV, E COLLECTING THE INFORMATION AND ENV
        SPREADING IT. EACH SYMBOL TABLE ENTRY HAS ONE COMPONENT,
        EITHER LABEL OR SYMBOL WHICH DEFINES THE KIND OF THE
        IDENTIFIER. THE ATTRIBUTE EMPTY HANDLES THE SYNTACTIC
        AMBIGUITY THAT ARISES WHEN THE FIRST STATEMENT IS EMPTY. THE
        PARSING IN WHICH THE LAST DECLARATION IS NOT EMPTY IS THE
        RIGHT ONE;

TERMINALS ARE . : ; ( ) '

RESERVED WORDS ARE TAPE, ALPHABET, IS, PRINT, MOVE, LEFT, RIGHT, ONE,
                   SQUARE, IF, THE, SYMBOL, THEN, GO, TO

ATTRIBUTES ARE

   DIRECTION = TITLE
   INDEX = INTEGER
   E = CONSTRUCT, CONSTRUCT
   ENV = E
```

```
   E1 = E
   E2 = E
   OBJPROG = LIST
   SP = TITLE
   MAP = CONSTRUCT, INTEGER
   INSTR = CONSTRUCT
   TAG = INTEGER
   LOC = INTEGER
   SYMBOL = INTEGER
   MOVE = TITLE
   LABEL = INTEGER
   P1 = POINTER
   P2 = P1
   M = INTEGER
   EMPTY = BOOLEAN

IDENTIFIERS ARE SIGMA WITH ATTRIBUTE SP

NONTERMINALS ARE

   P = S(OBJPROG)
   S = S(OBJPROG, E, EMPTY), I(ENV)
   L = S(OBJPROG, E, EMPTY), I(ENV)
   D = S(INDEX, E, EMPTY)
   O = S(DIRECTION)

START SYMBOL P

FORMATS ARE

  F1 = ("(", LOC, ": PRINT, ", SYMBOL, ")")
  F2 = ("(", LOC, ": MOVE, ", MOVE, ")")
  F3 = ("(", LOC, ": JUMP, ", LABEL,")")
  F4 = ("(", LOC, ": IF,", SYMBOL, ",", LABEL,")")
  F5 = ("(", LOC, ": STOP)")


FUNCTION JOINE (E1, E2);

BEGIN  COMMENT THIS PROCEDURE JOINS TWO SYMBOL TABLES AND  CHECKS FOR
         DUPLICATE ENTRIES;

   P1 := FIRST(E2);
   IF NULLB(E1) THEN E2 ELSE
   BEGIN
      WHILE ¬NULLB(P1) DO
      BEGIN
         IF ¬NULLR(FIND(E1,SELECTOR([P1]))) THEN
         ERROR(SELECTOR([P1]), " DECLARED TWICE");
         E1 := *[P1];  P1 := NEXT([P1])
      END;
```

```
      E1
   END
END;


PROCEDURE OUTPUT(OBJPROG);

BEGIN COMMENT THIS PROCEDURE TAKES THE OBJECT PROGRAM LIST AND PRINTS
         ITS INSTRUCTIONS.  IN THE PROCESS IT PLACES THE ADDRESS OF
         THE INSTRUCTION IN THE COMPONENT LOC AND BINDS LABELS TO
         ADDRESSES.  PSEUDO-INSTRUCTIONS (INSTRUCTION WITH COMPONENT
         TAG) ARE NOT PRINTED AND ARE USED TO BUILD THE MAP TABLE
         THAT GIVES THE CORRESPONDENCE BETWEEN VALUE-LABELS AND
         ADDRESS.  THE BINDING IS DONE BY SUBSTITUTING IN THE
         COMPONENT "LABEL" THE LABEL-VALUE BY THE ADDRESS ASSOCIATED
         WITH IT IN MAP.  THE BINDING IS DONE IN PARALLEL, USING THE
         PROCEDURE PLACE, SO THAT FORWARD REFERENCES CAN BE HANDLED
         WITHOUT WORRYING ABOUT PASSIVATIONS OCCURRING;

   M := 1;
   WHILE ¬NULLB(OBJPROG) DO
   BEGIN
      P1 := CAR(OBJPROG);   P2 := FIND([P1], TAG);
      IF NULLB(P2) THEN
      BEGIN COMMENT THIS IS AN INSTRUCTION. CHECK TO SEE IF THERE IS
                    A LABEL COMPONENT: IF THERE IS, RETRIEVE IT FROM
                    MAP AND ASSIGN IT;
         [P1].LOC := [M];   P2 := FIND([P1],LABEL);
         IF ¬NULLB(P2) THEN PLACE([P2],MAP);
         WRITE ([P1], /);   M := M +1;
      END ELSE
      COMMENT THIS IS A PSEUDO-INSTRUCTION. UPDATE MAP:
      MAP.[[P2]] := [M];
      OBJPROG := CDR(OBJPROG)
   END
END;


PROCEDURE PLACE(P2, MAP);

COMMENT THIS PROCEDURE WILL ASSIGN, IN PARALLEL, AN ADDRESS TO THE
```

COMPONENT LABEL. THIS WAY THE PROCEDURE OUTPUT IS REACTIVATED IMMEDIATELY, EVEN IF THIS IS A FORWARD JUMP. THE REASON A PROCEDURE IS CALLED INSTEAD OF JUST PLACING THE PARALLEL STATEMENT IN THE BODY OF THE CALLING PROGRAM IS THAT THE VALUE OF P2 WHEN THE CALL IS MADE HAS TO BE PRESERVED AND THE PROCEDURE PRODUCES A COPY OF IT. A PARALLEL PROCESS BY ITSELF DOES NOT PRODUCE NEW NODES AND AS THE VALUE OF P2 IS CONTINUALLY CHANGING THERE IS NO ASSURANCE (SINCE THE PROCESSES RUN ASYNCHRONOUSLY) THAT IT WOULD HAVE THE PROPER VALUE EVERY TIME;

```
$/ [P2] := MAP.[[P2]] /$
```

```
$P11  D ::= TAPE ALPHABET IS SIGMA
      $/ INDEX(D) := 1;   EMPTY(D) := FALSE;
         E(D).[SP(SIGMA)] := 1 /$


$P12  D ::= D ; SIGMA
      $/ EMPTY(D) := FALSE;  INDEX(D) := INDEX(D*) +1 /$
      $/ E(D) := JOINE(E(D*),
                       E *= $([SP(SIGMA)].SYMBOL := INDEX(D))) /$


$P13  D ::= D ;
      $/ EMPTY(D) := TRUE /$


$P21  S ::= PRINT ' SIGMA '
      $/ E(S) := NULL;
         OBJPROG(S) := LIST(INSTR *= $(FORMAT := F1;
                                SYMBOL := ENV(S).[SP(SIGMA)].SYMBOL)) /$
      $/ EMPTY(S) := FALSE /$


$P22  S ::= MOVE O ONE SQUARE
      $/ E(S) := NULL;
         OBJPROG(S) := LIST(INSTR *= $(FORMAT := F2;
                                       MOVE := DIRECTION(O))) /$
      $/ EMPTY(S) := FALSE /$


$P221 O ::= LEFT
      $/ DIRECTION(O) := "LEFT" /$


$P222 O ::= RIGHT
      $/ DIRECTION(O) := "RIGHT" /$


$P23  S ::= GO TO SIGMA
      $/ E(S) := NULL;
         OBJPROG(S) := LIST(INSTR *= $(FORMAT := F3;
                                  LABEL := ENV(S).[SP(SIGMA)].LABEL)) /$
      $/ EMPTY(S) := FALSE /$


$P24  S ::=
      $/ E(S) := NULL;  OBJPROG(S) := NULL /$
      $/ EMPTY(S) := TRUE /$


$P23  S ::= IF THE TAPE SYMBOL IS ' SIGMA ' THEN S
      $/ M := NEWINTEGER;
         OBJPROG(S) := CONS(INSTR *= $(FORMAT := F4;  LABEL := M;
                                 SYMBOL := ENV(S).[SP(SIGMA)].SYMBOL),
                            APEND(OBJPROG(S*),
                                  LIST(INSTR *= $(TAG := M)))) /$
      $/ EMPTY(S) := FALSE /$
```

```
$P32  S ::= SIGMA : S
      $/ M := NEWINTEGER;
         E(S) := JOINE(E(S*), E ** $([SP(SIGMA)].LABEL := M)) /$
      $/ OBJPROG(S) := CONS(INSTR ** $(TAG = M), OBJPROG(S*)) /$
      $/ EMPTY(S) := FALSE /$


$P33  S ::= ( L )
      $/ EMPTY(S) := FALSE /$


$P41  L ::= S


$P42  L ::= L ; S
      $/ E(L) := JOINE(E(L*),E(S)) /$
      $/ OBJPROG(L) := APEND(OBJPROG(L*), OBJPROG(S)) /$
      $/ EMPTY(L) := EMPTY(L*) /$


$P5   P ::= D ; L
      $/ OBJPROG(P) := APEND(OBJPROG(L),
                             LIST(INSTR ** $(FORMAT := F5)));
         OUTPUT(OBJPROG(P)) /$
      $/ ENV(L) := JOINE(E(D), E(L)) /$
      $/ IF EMPTY(D) THEN DAMB(FALSE, 1) ELSE
         IF EMPTY(L) THEN DAMB(TRUE, 1) /$
```

```
TAPE ALPHABET IS BLANK; UNO; ZERO; POINT;;
PRINT 'POINT';
GO TO CARRY;
TEST: IF THE TAPE SYMBOL IS 'UNO' THEN
      [PRINT 'ZERO';
       CARRY: MOVE LEFT ONE SQUARE; GO TO TEST];
PRINT 'UNO';
REALIGN: MOVE RIGHT ONE SQUARE;
         IF THE TAPE SYMBOL IS 'ZERO' THEN GO TO REALIGN.
```

PARSING TREE

LOCATION	AMBIGUOUS	BROTHER	SON	SEMANTICS	SELECTOR,(PRODUCTION OR VALUE)
0	46	0	1	1124	*P,$P5
1	0	37	2	1048	*L,$P42
2	0	6	3	766	*S,$P31
3	0	5	4	654	*S,$P23
4	0	0	0	4136	*SIGMA,REALIGN
5	0	6	0	5432	*SIGMA,ZERO
6	0	0	7	1048	*L,$P42
7	0	11	8	878	*S,$P32
8	0	10	9	564	*S,$P22
9	0	0	0	634	*O,$P222
10	0	0	0	4136	*SIGMA,REALIGN
11	0	0	12	1048	*L,$P42
12	0	14	13	492	*S,$P21
13	0	0	0	6824	*SIGMA,UNO
14	0	0	15	1048	*L,$P42
15	0	31	16	878	*S,$P32
16	0	30	17	766	*S,$P31

```
17      0       29      18      978     *S,$P33
18      0       0       19      1048    *L,$P42
19      0       21      20      654     *S,$P23
20      0       0       0       16064   *SIGMA,TEST
21      0       0       22      1048    *L,$P42
22      0       26      23      878     *S,$P32
23      0       25      24      564     *S,$P22
24      0       0       0       618     *D,$P22L
25      0       0       0       16040   *SIGMA,CARRY
26      0       0       27      1016    *L,$P41
27      0       0       28      492     *S,$P21
28      0       0       0       5432    *SIGMA,ZERO
29      0       0       0       6824    *SIGMA,UNO
30      0       0       0       16064   *SIGMA,TEST
31      0       0       32      1048    *L,$P42
32      0       34      33      654     *S,$P23
33      0       0       0       16040   *SIGMA,CARRY
34      0       0       35      1016    *L,$P41
35      0       0       36      492     *S,$P21
36      0       0       0       8664    *SIGMA,POINT
37      0       0       38      452     *D,$P13
38      0       0       39      384     *D,$P12
39      0       40      0       8664    *SIGMA,POINT
40      0       0       41      384     *D,$P12
41      0       42      0       5432    *SIGMA,ZERO
42      0       0       43      384     *D,$P12
43      0       44      0       6824    *SIGMA,UNO
44      0       0       45      336     *D,$P11
45      0       0       0       5784    *SIGMA,BLANK
46      0       0       47      1124    *P,$P5
47      0       61      48      1048    *L,$P42
48      0       49      3       766     *S,$P31
49      0       0       50      1048    *L,$P42
50      0       51      8       878     *S,$P32
51      0       0       52      1048    *L,$P42
52      0       53      13      492     *S,$P21
53      0       0       54      1048    *L,$P42
54      0       55      16      878     *S,$P32
55      0       0       56      1048    *L,$P42
56      0       57      33      654     *S,$P23
57      0       0       58      1048    *L,$P42
58      0       59      36      492     *S,$P21
59      0       0       60      1016    *L,$P41
60      0       0       0       718     *S,$P24
61      0       0       39      384     *D,$P12
```

APPENDIX 3

```
                101 *
                102 *
                103 *  THIS IS TURINGOL
                104 *
                105 *
4               106            DEF    T,.
4               107            DEF    T,:
4               108            DEF    T,;
4               109            DEF    T,(
4               110            DEF    T,)
4               111            DEF    T,'
                112 *
4               113            DEF    R,TAPE
4               114            DEF    R,ALPHABET
4               115            DEF    R,IS
4               116            DEF    R,PRINT
4               117            DEF    R,MOVE
4               118            DEF    R,LEFT
4               119            DEF    R,RIGHT
4               120            DEF    R,ONE
4               121            DEF    R,SQUARE
4               122            DEF    R,IF
4               123            DEF    R,THE
4               124            DEF    R,SYMBOL
4               125            DEF    R,THEN
4               126            DEF    R,GO
4               127            DEF    R,TO
                128 *
4               129            DEF    I,SIGMA,SP
                130 *
8               131            DEF    N,P,S(OBJPROG)
12              132            DEF    N,S,S(OBJPROG,E,EMPTY),1(ENV)
24              133            DEF    N,L,S(E,OBJPROG,EMPTY),1(ENV)
30              134            DEF    N,D,S(INDEX,E,EMPTY)
44              135            DEF    N,D,S(DIRECTION)
                136 *
48              137            DEF    S,P
                138 *
48              139            DEF    A,DIRECTION,TITLE
48              140            DEF    A,INDEX,INTEGER
48              141            DEF    A,E,<ENTRY>
54              142            DEF    A,ENTRY,<>
54              143            DEF    A,ENV,E
54              144            DEF    A,E1,E
54              145            DEF    A,E2,E
54              146            DEF    A,OBJPROG,LIST
54              147            DEF    A,SP,TITLE
54              148            DEF    A,MAP,<INTEGER>
60              149            DEF    A,INSTR,<>
```

```
00
00
00
00
00
00
00
00

00
00
00
00
00

100
100  13
101  48  5752
104  13
105  48  6200
108  47  5752
111  48 15352
114  47  6200
117  53     0/
120  61  5752 255
124  24
125  47 15352
128  54     0/
131  47  6200
134  47 15352
137  19
138  45     0
141  54     0/
144  29
145  47 15352
148  19
149  58  1104
152  31
153  15
154  61  6200 255
158  47 15352
161  20
162  47 15352
165  27
166  12
167  48 15352
170  52   128
173  61  6200 255
177  24
```

```
150          DEF   A,TAG,INTEGER
151          DEF   A,LOC,INTEGER
152          DEF   A,SYMBOL,INTEGER
153          DEF   A,MOVE,TITLE
154          DEF   A,LABEL,INTEGER
155          DEF   A,P1,POINTER
156          DEF   A,P2,P1
157          DEF   A,M,INTEGER
158          DEF   A,EMPTY,BOOLEAN
159 *
160          DEF   F,F1,("1",LOC,": PRINT,",SYMBOL,")")
161          DEF   F,F2,("1",LOC,": MOVE,",MOVE,")")
162          DEF   F,F3,("1",LOC,": JUMP,",LABEL,")")
163          DEF   F,F4,("1",LOC,": IF,",SYMBOL,",",LABEL,")")
164          DEF   F,F5,("1",LOC,": STOP)")
165 *
166 *
167 *
168 *
169 *  THIS PROCEDURE ATTACHES TWO SYMBOL TABLES AND CHECKS TO SEE THAT
170 *  THERE ARE NO NAME CONFLICTS
171 *
172          PROC  JOINE
173          FLIP
174          STO   E2
175          FLIP
176          STO   E1
177          LOAD  E2
178          STO   P1
179          LOAD  E1
180          JUMPT EVLA
181          GET   E2,LOCAL
182          RET
183 EVLA     LOAD  P1
184 EVAL     JUMPF TERM
185          LOAD  E1
186          LOAD  P1
187          NAME
188          FIND
189          JUMPF GOON
190          ERROR
191          LOAD  P1
192          NAME
193          VALC  "DECLARED TWICE"
194          OUTF
195          MLT
196 GOON     GET   E1,LOCAL
197          LOAD  P1
198          REP
199          LOAD  P1
200          GETN
201          DBL
202          STO   P1
203          JUMP  EVAL
204 TERM     GET   E1,LOCAL
205          RET
206 *
207 *  THIS PROCEDURE BINDS LABELS WITH ADDRESSES AND OUTPUTS THE
208 *  INSTRUCTIONS OF OBJPROG
209 *
```

```
178
178  13
179  48  8384
182  59  2024
185  48  4680
188  47  8384
191  54     0/
194  61  8384 255
198   7
199  12
200  48 15352
203  28
204  45  3768
207  12
208  48  2528
211  53     0/
214  47 15352
217  60 11+60    0
221  47  4680
224   5
225  47 15352
228  28
229  45  4720
232  12
233  48  2528
236  54     0/
239  47  2528
242  61 16160 255
246  57     0/
249  47 15352
252  31
253  33
254  47  4680
257  36     2
259  48  4680
262  52     0/
265  60 16160 255
269  47  2528
272  28
273  60     0    0
277  47  4680
280   5
281  61  8384 255
285   8
286  12
287  48  8384
290  53   194
293  24


294
294  13
295  48 16160
298  13
299  48  2528
302  50     0/
305  24
306  47  2528
309  61 16160 255

210          PROC  OUTPUT
211          FLIP
212          STO   OBJPROG
213          VALC  1
214          STO   M
215          LOAD  OBJPROG
216          JUMPF FINISH
217 BEGIN    GET   OBJPROG,LOCAL
218          CAR
219          DBL
220          STO   P1
221          VAL
222          FIND  TAG
223          DBL
224          STO   P2
225          JUMPT ELSE
226          LOAD  P1
227          PLA   LOC
228          LOAD  M
229          ASS
230          LOAD  P1
231          VAL
232          FIND  LABEL
233          DBL
234          STO   P2
235          JUMPF WRITE
236          LOAD  P2
237          GET   MAP,LOCAL
238          CALL  PLACE
239 WRITE    LOAD  P1
240          OUTF
241          OUTC
242          LOAD  M
243          AR    ONEP
244          STO   M
245          JUMP  NEXT
246 ELSE     PLA   MAP,LOCAL
247          LOAD  P2
248          VAL
249          PLA
250          LOAD  M
251          ASS
252 NEXT     GET   OBJPROG,LOCAL
253          CDR
254          DBL
255          STO   OBJPROG
256          JUMPT BEGIN
257 FINISH   RET
258 *
259 *  THIS PROCEDURE ACTUALLY ASSIGN THE ADDRESS TO THE LABEL
260 *
261          PROC  PLACE
262          FLIP
263          STO   MAP
264          FLIP
265          STO   P2
266          PARN  A1
267          RET
268 A1       LOAD  P2
269          GET   MAP,LOCAL
```

```
313  47  2528                 270          LOAD  P2
316  28                       271          VAL
317  61     0   0              272          GET
321   5                       273          ASS
322  15                       274          HLT
                              275 *
                              276 *  THE PRODUCTIONS OF TURINGOL
                              277 *
323                           278          PROD  D,TAPE,ALPHABET,IS,SIGMA
336                           279          LABEL P11
338  60 11960   1              280          PLA   INDEX,D
342  58  2624                  281          ASSI  1
345  60  3960   1              282          PLA   EMPTY,D
349  58  7456                  283          ASSI  FALSE
352  60  2512   1              284          PLA   E,D
356  61  8736   2              285          GET   SP,SIGMA
360  28                       286          VAL
361  60     0   0              287          PLA
365  60 11680   0              288          PLA   SYMBOL
369  58  2624                  289          ASSI  1
372  15                       290          HLT
373                           291          ENDP
                              292 *
                              293 *
373                           294          PROD  D,D,:,SIGMA
384                           295          LABEL P12
386  56     0/                 296          PARN  X1
389  60  3960   1              297          PLA   EMPTY,D
393  58  7456                  298          ASSI  FALSE
396  60 11960   1              299          PLA   INDEX,D
400  61 11960   3              300          GET   INDEX,D*
404  28                       301          VAL
405  36     2                  302          AR    ONEP
407   5                       303          ASS
408  15                       304          HLT
409  60  2512   1              305 X1       PLA   E,D
413  61  2512   3              306          GET   E,D*
417  51  6200                  307          PLAN  E1
420  12                       308          DBL
421  61  8736   2              309          GET   SP,SIGMA
425  28                       310          VAL
426  60     0   0              311          PLA
430  60 11680   0              312          PLA   SYMBOL
434  61 11960   1              313          GET   INDEX,D
438   5                       314          ASS
439  57   100                  315          CALL  JOINE
442   5                       316          ASS
443  15                       317          HLT
444                           318          ENDP
                              319 *
                              320 *
444                           321          PROD  D,D,:
452                           322          LABEL P13
454  60  3960   1              323          PLA   EMPTY,D
458  58  2912                  324          ASSI  TRUE
461  15                       325          HLT
462                           326          ENDP
         CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
462  63   2
        11960   1   2 INDEX
```

```
343  47  2528
346  28
347  61     0   0
351   5
352  15


353
356
358  60 11960   1
362  58  2624
365  60  3960   1
369  58  1456
372  60  2512   1
376  61  8736   2
380  28
381  60     0   0
385  60 11680   0
389  58  2624
392  15
393


393
394
396  56     0/
399  60  3960   1
403  58  7456
406  60 11960   1
410  61 11960   3
414  28
415  36     2
417   5
418  15
419  60  2512   1
423  61  2512   3
427  51  0266
430  12
431  61  8736   2
435  26
436  60     0   0
440  60 11680   0
444  61 11960   1
448   5
449  57   100
452   5
453  15
454


454
462
466  60  3960   1
470  58  2912
473  15
474
```

```
270          LOAD  P2
271          VAL
272          GET
273          ASS
274          MLT
275 *
276 *  THE PRODUCTIONS OF TURINGOL
277 *
278          PROD  D,TAPE,ALPHABET,IS,SIGMA
279          LABEL P11
280          PLA   INDEX,D
281          ASSI  1
282          PLA   EMPTY,D
283          ASSI  FALSE
284          PLA   E,D
285          GET   SP,SIGMA
286          VAL
287          PLA
288          PLA   SYMBOL
289          ASSI  1
290          MLT
291          ENDP
292 *
293 *
294          PROD  D,D,:,SIGMA
295          LABEL P12
296          PARM  X1
297          PLA   EMPTY,D
298          ASSI  FALSE
299          PLA   INDEX,D
300          GET   INDEX,D*
301          VAL
302          AR    ONEP
303          ASS
304          MLT
305 X1       PLA   E,D
306          GET   E,D*
307          PLAN  E1
308          DBL
309          GET   SP,SIGMA
310          VAL
311          PLA
312          PLA   SYMBOL
313          GET   INDEX,D
314          ASS
315          CALL  JGINE
316          ASS
317          MLT
318          ENDP
319 *
320 *
321          PROD  D,D,;
322          LABEL P13
323          PLA   EMPTY,D
324          ASSI  TRUE
325          MLT
326          ENDP
```

```
      CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
462  63   2
       11960   1   2 INDEX
```

```
          2512   1   2 E
472  15
473  56   462
476  52   454
                                 *
                                 *
479                                      PROD  S,PRINT,',SIGMA,'
492                                      LABEL P21
494  56     0/                           PARN  X1
497  60  2512   1                        PLA   E,S
501  58 14752                            ASSI  NIL
504  60  8384   1                        PLA   OBJPROG,S
508  49                                  LIST
511  51  7664                            PLAN  INSTR
514  12                                  DBL
515  50 13952                            FMT   F1
518  12                                  DBL
519  60 11680   0                        PLA   SYMBOL
523  61 13776   1                        GET   FNV,S
527  61  8736   2                        GET   SP,SIGMA
531  28                                  VAL
532  61     0   0                        GET
536  61 11680   0                        GET   SYMBOL
540   5                                  ASS
541  10                                  CONS
542   5                                  ASS
543  15                                  HLT
544  60  3960   1                  X1    PLA   EMPTY,S
548  58  7456                            ASSI  FALSE
551  15                                  HLT
552                                      ENDP
                                 *
                                 *
552                                      PROD  S,MOVE,O,ONE,SQUARE
564                                      LABEL P22
566  56     C/                           PARN  X1
569  60  2512   1                        PLA   E,S
573  58 14752                            ASSI  NIL
576  60  8384   1                        PLA   OBJPROG,S
580  49                                  LIST
583  51  7664                            PLAN  INSTR
586  12                                  DBL
587  50  1160                            FMT   F2
590  12                                  DBL
591  60  4448   0                        PLA   MOVE
595  61 15240   2                        GET   DIRECTION,O
599   5                                  ASS
600  10                                  CONS
601   5                                  ASS
602  15                                  HLT
603  60  3960   1                  X1    PLA   EMPTY,S
607  58  7456                            ASSI  FALSE
610  15                                  HLT
611                                      ENDP
                                 *
                                 *
611                                      PROD  O,LEFT
618                                      LABEL P221
620  60 15240   1                        PLA   DIRECTION,O
624  58  2728                            ASSI  "LEFT"
```

```
627  15
628
628
634
636  60 15240   1
640  58  2976
643  15
644
644
654
656  56     0/
659  60  2512   1
663  59 14752
666  60  8384   1
670  49
673  51  7664
676  12
677  50  4712
680  12
681  60  4720   0
685  61 13776   1
689  61  8736   2
693  28
694  61     0   0
698  61  4720   0
702   5
703  10
704   5
705  15
706  60  3960   1
710  58  7456
713  15
714
714
718
720  60  2512   1
724  58 14752
727  60  8384   1
731  58 14752
734  60  3960   1
738  58  2912
741  15
742
742
766
768  56     0/
771  14
772  48  4680
775  60  8384   1
779  61  8384   2
783  49
786  51  7664
```

```
383          HLT
384          ENDP
385 *
386 *
387          PROD  O,RIGHT
388          LABEL P222
389          PLA   DIRECTION,O
390          ASSI  "RIGHT"
391          HLT
392          ENDP
393 *
394 *
395          PROD  S,GO,TO,SIGMA
396          LABEL P23
397          PARM  X1
398          PLA   E,S
399          ASSI  NIL
400          PLA   OBJPROG,S
401          LIST
402          PLAN  INSTR
403          DBL
404          FMT   F3
405          DBL
406          PLA   LABEL
407          GET   ENV,S
408          GET   SP,SIGMA
409          VAL
410          GET
411          GET   LABEL
412          ASS
413          CONS
414          ASS
415          HLT
416 X1       PLA   EMPTY,S
417          ASSI  FALSE
418          HLT
419          ENDP
420 *
421 *
422          PROD  S
423          LABEL P24
424          PLA   E,S
425          ASSI  NIL
426          PLA   OBJPROG,S
427          ASSI  NIL
428          PLA   EMPTY,S
429          ASSI  TRUE
430          HLT
431          ENDP
432 *
433 *
434          PROD  S,IF,THE,TAPE,SYMBOL,IS,',SIGMA,',THEN,S
435          LABEL P31
436          PARM  X1
437          GEN
438          STO   M
439          PLA   OBJPROG,S
440          GET   OBJPROG,S*
441          LIST
442          PLAN  INSTR
```

```
789  12                          443          DBL
790  60  3768   0                444          PLA   TAG
794  47  4680                    445          LOAD  M
797   5                          446          ASS
798  10                          447          CONS
799  23                          448          APEND
800  51  7664                    449          PLAN  INSTR
803  12                          450          DBL
804  50  8264                    451          FMT   F4
807  12                          452          DBL
808  60  4720   0                453          PLA   LABEL
812  47  4680                    454          LOAD  M
815   5                          455          ASS
816  12                          456          DBL
817  60 11680   0                457          PLA   SYMBOL
821  61 13776   1                458          GET   ENV,S
825  61  8736   3                459          GET   SP,SIGMA
829  28                          460          VAL
830  61     0   0                461          GET
834  61 11680   0                462          GET   SYMBOL
838   5                          463          ASS
839  10                          464          CONS
840   5                          465          ASS
841  15                          466          HLT
842  60  3960   1                467 X1       PLA   EMPTY,S
846  58  7456                    468          ASSI  FALSE
849  15                          469          HLT
850                              470          ENDP
         CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
850  63   2
        13776   2   1 ENV
         2512   1   2 E
860  15
861  56   850
864  52   768
                                 471 *
                                 472 *
867                              473          PROD  S,SIGMA,:,S
878                              474          LABEL P32
880  56     0/                   475          PARN  X1
883  56     0/                   476          PARN  X2
886  14                          477          GEN
887  48  4680                    478          STO   M
890  60  2512   1                479          PLA   E,S
894  61  2512   2                480          GET   E,S*
898  51  6200                    481          PLAN  E1
901  12                          482          DBL
902  61  8736   3                483          GET   SP,SIGMA
906  28                          484          VAL
907  60     0   0                485          PLA
911  60  4720   0                486          PLA   LABEL
915  47  4680                    487          LOAD  M
918   5                          488          ASS
919  57   100                    489          CALL  JOINE
922   5                          490          ASS
923  15                          491          HLT
924  60  8384   1                492 X1       PLA   OBJPROG,S
928  61  8384   2                493          GET   OBJPROG,S*
932  51  7664                    494          PLAN  INSTR
935  12                          495          DBL
```

```
936  60  3768   0          496          PLA   TAG
940  47  4680              497          LOAD  M
943   5                    498          ASS
944  10                    499          CONS
945   5                    500          ASS
946  15                    501          HLT
947  60  3960   1          502  X2      PLA   EMPTY,S
951  58  7456              503          ASSI  FALSE
954  15                    504          HLT
955                        505          ENDP
        CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
955  63   1
       13776   2   1 ENV
961  15
962  56   955
965  52   880
                           506 *
                           507 *
968                        508          PROD  S,I,L,I
978                        509          LABEL P33
980  60  3960   1          510          PLA   EMPTY,S
984  58  7456              511          ASSI  FALSE
987  15                    512          HLT
988                        513          ENDP
        CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
988  63   3
       13776   2   1 ENV
        8384   1   2 OBJPROG
        2512   1   2 E
1002 15
1003 56   988
1006 52   980
                           514 *
                           515 *
1009                       516          PROD  L,S
1016                       517          LABEL P41
1018                       518          ENDP
        CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
1018 63   4
       13776   2   1 ENV
        8384   1   2 OBJPROG
        3960   1   2 EMPTY
        2512   1   2 E
1036 15
                           519 *
                           520 *
1037                       521          PROD  L,L,;,S
1046                       522          LABEL P42
1050 56     07             523          PARM  X1
1053 60  3960   1          524          PLA   EMPTY,L
1057 61  3960   3          525          GET   EMPTY,L*
1061  5                    526          ASS
1062 60  2512   1          527          PLA   E,L
1066 61  2512   3          528          GET   E,L*
1070 61  2512   2          529          GET   E,S
1074 57   100               530          CALL  JOINE
1077  5                    531          ASS
1078 15                    532          HLT
1079 60  8384   1          533 X1       PLA   OBJPROG,L
1083 61  8384   3          534          GET   OBJPROG,L*
```

```
1087  61  8384    2          535           GET   OBJPROG,S
1091  23                     536           APEND
1092   5                     537           ASS
1093  15                     538           HLT
1094                         539           ENDP
          CODE GENERATED FOR THE IMPLIED SEMANTIC RULES
1094  63   2
          13776    3    1 ENV
          13776    2    1 ENV
1104  15
1105  56  1094
1108  92  1050
                             540 *
                             541 *
1111                         542           PROD  P,0,1,L,,
1124                         543           LABEL P5
1125  56     0/              544           PARN  X1
1129  56     0/              545           PARN  X2
1132  61  3960    3          546           GET   EMPTY,0
1136  28                     547           VAL
1137  54     0/              548           JUMPF TR
1140  59  7456               549           VALC  FALSE
1143  44 11672               550           DAMB  P
1146  61  3960    2          551 TR        GET   EMPTY,L
1150  28                     552           VAL
1151  54     0/              553           JUMPF EXIT
1154  59  2912               554           VALC  TRUE
1157  44 11672               555           DAMB  P
1160  15                     556 EXIT      HLT
1161  60  8384    1          557 X1        PLA   OBJPROG,P
1165  12                     558           DBL
1166  61  8384    2          559           GET   OBJPROG,L
1170  49                     560           LIST
1173  51  7664               561           PLAN  INSTR
1176  12                     562           DBL
1177  50 11868               563           FMT   FS
1180  10                     564           CONS
1181  23                     565           APEND
1182   5                     566           ASS
1183  57   178               567           CALL  OUTPUT
1186  15                     568           HLT
1187  60 13776    2          569 X2        PLA   ENV,L
1191  61  2512    2          570           GET   E,L
1195  61  2512    3          571           GET   E,0
1199  57   100               572           CALL  JOINE
1202   5                     573           ASS
1203  15                     574           HLT
1204                         575           ENDP
1204                         576           END

NO ERRORS DETECTED

# OF INSTRUCTION CARDS PROCESSED=  518
```

APPENDIX L

(I)

```
COMMENT  PROGRAMS (I) THROUGH (VIII) ARE INTENDED AS EXAMPLES OF HOW
         DIFFERENT SIMULA FEATURES TRANSLATE INTO THE TARGET LANGUAGE
         AND NOT AS ACTUAL WORKING PROGRAMS. PROGRAMS (VIIII) THROUGH
         (X) ARE TAKEN FROM THE LITERATURE.
         THE PARSE TREES GENERATED FOR PROGRAMS (I) AND (III) ARE
         INCLUDED IN THEIR RESPECTIVE OUTPUTS;

BEGIN INTEGER I; I := 4 END
```

PARSING TREE

LOCATION	AMBIGUOUS	BROTHER	SON	SEMANTICS	SELECTOR(PRODUCTION OR VALUE)
0	0	0	1	12668	*PROGRAM,$P184
1	0	0	2	13386	*BLOCK,$P174
2	0	0	3	14056	*UNLBLOCK,$P179
3	0	26	4	22926	*COMPT,$P264
4	0	0	5	23204	*ST,$P265A
5	0	0	6	23300	*ST1,$P266
6	0	0	7	23660	*UNCONDST,$P270
7	0	0	8	23910	*BASICST,$P273
8	0	0	9	24282	*UNLBASICST,$P275
9	0	0	10	24650	*ASSST,$P283
10	0	0	11	24746	*VALASS,$P285
11	0	19	12	25104	*VALRPART,$P288
12	0	0	13	5034	*VALEXPR,$P4
13	0	0	14	5122	*ARITEXPR,$P7
14	0	0	15	5332	*SARITEXPR,$P9
15	0	0	16	5666	*TERM,$P14
16	0	0	17	5906	*FAC,$P19
17	0	0	18	6146	*PRIM,$P21
18	0	0	0	5376	*NO,4
19	22	0	20	24946	*VALLPART,$P287
20	0	0	21	25048	*PROCID2,$P287A
21	0	0	0	4160	*SIGMA,I
22	0	0	23	24864	*VALLPART,$P286
23	0	0	24	6464	*VAR,$P48
24	0	0	25	6652	*IDI,$P52
25	0	0	0	4160	*SIGMA,I
26	0	0	27	15212	*BLOCKHEAD,$P184
27	0	0	28	15554	*DECL,$P186
28	0	0	29	15988	*TYPEDECL,$P192
29	0	31	30	16400	*TYPELIST,$P202
30	0	0	0	4160	*SIGMA,I
31	0	0	32	16128	*TYPEN,$P193

```
    32          0          0          0       16192       *VALTYPE;SP196


ORIGIN=     1

1
C(INTEGER)
ADA(ADDR=(LM=4,CM=1))
C(INTEGER(VALUE=4))
STO(FALSE)
RET


 RALPH JOB STATISTICS --  0.23 MINUTES EXECUTION TIME
                          0.03 MINUTES CPU TIME   0.20 MINUTES WAIT TIME
```

```
COMMENT IN THIS EXAMPLE IT IS INTERESTING TO OBSERVE THE NUMBER OF
        AMBIGUITIES GENERATED BY AN ACTUAL PARAMETER WHICH IS A
        SINGLE IDENTIFIER;

BEGIN
   INTEGER I,J;
   REAL PROCEDURE P(K); INTEGER K;  P := K**2;
   J := 2;  I := P(J)
END
```

PARSING TREE

LOCATION	AMBIGUOUS	BROTHER	SON	SEMANTICS	SELECTOR(PRODUCTION OR VALUE)
0	0	0	1	12668	*PROGRAM,$P169
1	0	0	2	13336	*BLOCK,$P174
2	0	0	3	14056	*UNLBLOCK,$P179
3	0	110	4	23004	*CCMPT,$P265
4	0	88	5	22926	*CCMPT,$P264
5	0	0	6	23204	*ST,$P265A
6	0	0	7	23300	*STI,$P266
7	0	0	8	23660	*UNCONDST,$P270
8	0	0	9	23910	*BASICST,$P273
9	0	0	10	24282	*UNLBASICST,$P275
10	0	0	11	24650	*ASSST,$P283
11	0	0	12	24746	*VALASS,$P285
12	0	81	13	25104	*VALRPART,$P288
13	72	0	14	5034	*VALEXPR,$P4
14	0	0	15	5122	*ARITEXPR,$P7
15	0	0	16	5352	*SARITEXPR,$P9
16	0	0	17	5666	*TERM,$P14
17	0	0	18	5986	*FAC,$P19
18	0	0	19	6304	*PRIM,$P23
19	0	0	20	7470	*FUNC,$P89
20	0	69	21	7736	*APPART,$P92
21	0	0	22	7772	*APLIST,$P93
22	26	0	23	8016	*AP,$P96
23	0	0	24	7168	*ARID1,$P85
24	0	0	25	6652	*ID1,$P52
25	0	0	0	608	*SIGMA,J
26	29	0	27	8142	*AP,$P97
27	0	0	28	12510	*SWID1,$P168
28	0	0	0	608	*SIGMA,J
29	32	0	30	8224	*AP,$P98
30	0	0	31	7620	*PROCID1,$P90
31	0	0	25	6652	*ID1,$P52
32	0	0	33	7902	*AP,$P95
33	57	0	34	4890	*EXPR,$P1

34	46	0	35	5034	*VALEXPR,$P4
35	0	0	36	5122	*ARITEXPR,$P7
36	0	0	37	5382	*SARITEXPR,$P9
37	0	0	38	5666	*TERM,$P14
38	0	0	39	5986	*FAC,$P19
39	42	0	40	6200	*PRIM,$P22
40	0	0	41	6464	*VAR,$P48
41	0	0	25	6652	*IDI,$P52
42	0	0	43	6304	*PRIM,$P23
43	0	0	44	7470	*FUNC,$P89
44	0	45	0	7708	*APPART,$P91
45	0	0	31	7620	*PROCIDI,$P90
46	0	0	47	5078	*VALEXPR,$P5
47	0	0	48	8350	*BEXPR,$P103
48	0	0	49	8540	*SBOOL,$P105
49	0	0	50	8708	*IMPL,$P107
50	0	0	51	8876	*BTERM,$P109
51	0	0	52	9044	*BFAC,$P111
52	0	0	53	9212	*BSEC,$P113
53	55	0	54	9386	*BPRIM,$P116
54	0	0	41	6464	*VAR,$P48
55	0	0	56	9474	*BPRIM,$P117
56	0	0	44	7470	*FUNC,$P89
57	62	0	58	4978	*EXPR,$P3
58	0	0	59	11976	*DESIGEXPR,$P161
59	0	0	60	12142	*SDESIGEXPR,$P163
60	0	0	61	12252	*LABEL1,$P166
61	0	0	0	608	*SIGMA,J
62	0	0	63	4934	*EXPR,$P2
63	0	0	64	10360	*REFEXPR,$P147
64	0	0	65	10404	*OBJEXPR,$P148
65	67	0	66	10734	*SUBJEXPR,$P151
66	0	0	41	6464	*VAR,$P48
67	0	0	68	11030	*SOBJEXPR,$P152
68	0	0	44	7470	*FUNC,$P89
69	0	0	70	7620	*PROCIDI,$P90
70	0	0	71	6652	*IDI,$P52
71	0	0	0	6104	*SIGMA,P
72	0	0	73	5078	*VALEXPR,$P5
73	0	0	74	8350	*BEXPR,$P103
74	0	0	75	8540	*SBOOL,$P105
75	0	0	76	8708	*IMPL,$P107
76	0	0	77	8876	*BTERM,$P109
77	0	0	78	9044	*BFAC,$P111
78	0	0	79	9212	*BSEC,$P113
79	0	0	80	9474	*BPRIM,$P117
80	0	0	20	7470	*FUNC,$P89
81	84	0	82	24946	*VALLPART,$P287
82	0	0	83	25048	*PROCIDZ,$P287A
83	0	0	0	4160	*SIGMA,I
84	0	0	85	24804	*VALLPART,$P286
85	0	0	86	6464	*VAR,$P48
86	0	0	87	6652	*IDI,$P52
87	0	0	0	4160	*SIGMA,I
88	0	0	89	23204	*ST,$P265A
89	0	0	90	23300	*STI,$P266
90	0	0	91	23660	*UNCONST,$P270
91	0	0	92	23910	*BASICST,$P273
92	0	0	93	24282	*UNLBASICST,$P275
93	0	0	94	24650	*ASSST,$P283

94	0	0	95	24746	*VALASS,$P285
95	0	103	96	25104	*VALRPART,$P288
96	0	0	97	5034	*VALEXPR,$P4
97	0	0	98	5122	*ARITEXPR,$P7
98	0	0	99	5332	*SARITEXPR,$P9
99	0	0	100	5666	*TERM,$P14
100	0	0	101	5986	*FAC,$P19
101	0	0	102	6146	*PRIM,$P21
102	0	0	0	14480	*NU,2
103	106	0	104	24946	*VALLPART,$P287
104	0	0	105	25048	*PROCID2,$P287A
105	0	0	0	608	*SIGMA,J
106	0	0	107	24864	*VALLPART,$P286
107	0	0	108	6464	*VAR,$P48
108	0	0	109	6652	*IDI,$P52
109	0	0	0	608	*SIGMA,J
110	0	0	111	15306	*BLOCKHEAD,$P185
111	0	168	112	15778	*DECL,$P184
112	0	0	113	18268	*PROCDECL,$P221
113	0	145	114	20302	*PROCBODY,$P247
114	0	0	115	23300	*STI,$P266
115	0	0	116	23660	*UNCLNDST,$P270
116	0	0	117	23910	*BASICST,$P273
117	0	0	118	24282	*CALBAS.CST,$P275
118	0	0	119	24650	*ASSST,$P283
119	0	0	120	24746	*VALASS,$P285
120	0	138	121	25104	*VALRPART,$P288
121	0	0	122	5034	*VALEXPR,$P4
122	0	0	123	5122	*ARITEXPR,$P7
123	0	0	124	5332	*SARITEXPR,$P9
124	0	0	125	5666	*TERM,$P14
125	0	0	126	6034	*FAC,$P20
126	0	128	127	6146	*PRIM,$P21
127	0	0	0	14480	*NU,2
128	0	0	129	5986	*FAC,$P19
129	133	0	130	6200	*PRIM,$P22
130	0	0	131	6464	*VAR,$P48
131	0	0	132	6652	*IDI,$P52
132	0	0	0	12432	*SIGMA,K
133	0	0	134	6304	*PRIM,$P23
134	0	0	135	7470	*FUNC,$P89
135	0	136	0	7703	*APPART,$P91
136	0	0	137	7620	*PROCID1,$P90
137	0	0	132	6652	*IDI,$P52
138	141	0	139	24946	*VALLPART,$P287
139	0	0	140	25048	*PROCID2,$P287A
140	0	0	0	6104	*SIGMA,P
141	0	0	142	24864	*VALLPART,$P286
142	0	0	143	6464	*VAR,$P48
143	0	0	144	6652	*IDI,$P52
144	0	0	0	6104	*SIGMA,P
145	0	165	146	18770	*PROCHEAD,$P223
146	0	153	147	20310	*SPPART,$P239
147	0	149	148	19398	*IDLIST,$P234
148	0	0	0	12432	*SIGMA,K
149	0	152	150	20602	*SPECIFIER,$P240
150	0	0	151	16128	*TYPEN,$P193
151	0	0	0	16192	*VALTYPE,$P196
152	0	0	0	20266	*SPPART,$P238
153	156	159	154	19096	*MUPART,$P230

154	0	155	0	20242	*NAMEPART,$P237
155	0	0	0	19372	*VALPART,$P233
156	0	159	157	19194	*MOPART,$P231
157	0	158	0	19372	*VALPART,$P233
158	0	0	0	20242	*NAMEPART,$P237
159	0	163	160	18946	*FPPART,$P226
160	0	0	161	18970	*FPLIST,$P227
161	0	0	162	19078	*FP,$P229
162	0	0	0	12432	*SIGMA,K
163	0	0	164	18904	*PROCID,$P224
164	0	0	0	6104	*SIGMA,P
165	0	0	166	18724	*TYPEP,$P221A
166	0	0	167	16128	*TYPEN,$P193
167	0	0	0	16104	*VALTYPE,$P195
168	0	0	169	15212	*BLOCKHEAD,$P184
169	0	0	170	15554	*DECL,$P186
170	0	0	171	15968	*TYPEDECL,$P192
171	0	175	172	16480	*TYPELIST,$P203
172	0	174	173	16400	*TYPELIST,$P202
173	0	0	0	608	*SIGMA,J
174	0	0	0	4160	*SIGMA,I
175	0	0	176	16128	*TYPEN,$P193
176	0	0	0	16192	*VALTYPE,$P196

```
ORIGIN=    1

3
CHEID=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=FALSE)
ADR(ADDR=(LN=5,DN=1))
VAL(ADDR=(LN=5,DN=2))
C(INTEGER(VALUE=2))
AR(**)
STO(FALSE)
RET

1
C(INTEGER)
C(INTEGER)
C(PROCEDURE(LEVEL=5,SEGMENT=3))
ADR(ADDR=(LN=4,DN=2))
C(INTEGER(VALUE=2))
STO(FALSE)
ADR(ADDR=(LN=4,DN=1))
MARK
C(RET)
C(REAL)
C(ACTUAL(BODY=7,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=4,DN=3))
ENT
STO(FALSE)
RET

7
ADR(ADDR=(LN=4,DN=2))
RET
```

```
RALPH JOB STATISTICS --  0.20 MINUTES EXECUTION TIME
                        0.05 MINUTES CPU TIME   0.16 MINUTES WAIT TIME
```

```
SARA:  ISU:
BEGIN
   INTEGER CLARA;
   BOOLEAN NOISES;
   NOISES := TRUE;
   CLARA := IF NOISES THEN 1 ELSE 2;
   IF NOISES THEN GO TO MANTZE ELSE GO TO SARA;
   MANTZE:  GO TO ISU
END
```

```
ORIGIN=     1

1
C(LABEL(SEGMENT=1,DISP=3))
C(LABEL(SEGMENT=1,DISP=3))
MARK
ENT(LEVEL=3,BODY=4)
RET

4
C(INTEGER)
C(BOOLEAN)
C(LABEL(SEGMENT=4,DISP=21))
ADR(ADDR=(LN=5,CN=2))
C(BOOLEAN(VALUE="TRUE"))
STO(FALSE)
ADR(ADDR=(LN=5,CN=1))
VAL(ADDR=(LN=5,CN=2))
IFJ(12)
C(INTEGER(VALUE=1))
GO(13)
C(INTEGER(VALUE=2))
STO(FALSE)
VAL(ADDR=(LN=5,CN=2))
IFJ(19)
VAL(ADDR=(LN=5,CN=3))
GO
GO(21)
VAL(ADDR=(LN=4,CN=1))
GO
VAL(ADDR=(LN=4,CN=2))
GO
RET
```

```
 RALPH JOB STATISTICS --  0.19 MINUTES EXECUTION TIME
                          0.05 MINUTES CPU TIME   0.15 MINUTES WAIT TIME
```

```
COMMENT  HERE AND IN THE FOLLOWING PROGRAMS @( AND )@ ARE USED TO
         REPLACE SQUARE BRACKETS;

BEGIN
   PROCEDURE P(L,K);  NAME L;  VALUE K;  INTEGER L,K;
   BEGIN
      INTEGER ARRAY BRUNO,BETO@(1:L,1:2)@,SARITA@(K:K+L)@;
      MOISES := L;
      WHILE MOISES > 0 DO
      BEGIN
         BRUNO@(MOISES,1)@ := BETO@(MOISES,2)@ := 0;
         SARITA@(K+MOISES)@ := 1;
         MOISES := MOISES - 1;
      END
   END;
   REAL MOISES;
   P(3,5)
END
```



```
ORIGIN=     1

3
CHE(L=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=NAME,CLASS=FALSE)
CHE(D=3,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=FALSE)
C(INTEGER(VALUE=1))
VAL(ADDR=(LN=5,DN=2))
C(INTEGER(VALUE=1))
C(INTEGER(VALUE=2))
MAK(GENUS=(TYPE=INTEGER    KIND=ARRAY),N=2,COPIES=1)
VAL(ADDR=(LN=5,DN=3))
VAL(ADDR=(LN=5,DN=3))
VAL(ADDR=(LN=5,DN=2))
AR(0)
MAK(GENUS=(TYPE=INTEGER    KIND=ARRAY),N=1,COPIES=0)
ADR(ADDR=(LN=4,DN=2))
VAL(ADDR=(LN=5,DN=2))
STO(FALSE)
VAL(ADDR=(LN=4,DN=2))
C(INTEGER(VALUE=0))
COMP(>)
BFJ(46)
VAL(ADDR=(LN=5,DN=4))
VAL(ADDR=(LN=4,DN=2))
INX(VALUE)
C(INTEGER(VALUE=1))
INX(ADDR)
```

```
VAL(ADDR=(LN=5,DN=5))
VAL(ADDR=(LN=4,DN=2))
INX(VALUE)
C(INTEGER(VALUE=2))
INX(ADDR)
C(INTEGER(VALUE=0))
STO(TRUE )
STO(FALSE)
VAL(ADDR=(LN=5,DN=6))
VAL(ADDR=(LN=5,DN=3))
VAL(ADDR=(LN=4,DN=2))
AR(+)
INX(ADDR)
C(INTEGER(VALUE=1))
STO(FALSE)
ACR(ADDR=(LN=4,DN=2))
VAL(ADDR=(LN=4,DN=2))
C(INTEGER(VALUE=1))
AR(-)
STO(FALSE)
GOTO(6)
RET

7
C(INTEGER(VALUE=3))
RET

8
C(INTEGER(VALUE=5))
RET

1
C(PROCEDURE(LEVEL=5,SEGMENT=3))
C(REAL)
MARK
C(RET)
C(INTEGER)
C(ACTUAL(BODY=7,LEVEL=5,QUAL=,UNCERTYPE=INTEGER))
C(ACTUAL(BODY=8,LEVEL=5,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=4,DN=1))
ENT
DEL
RET


  RALPH JOB STATISTICS --  0.24 MINUTES EXECUTION TIME
                           0.07 MINUTES CPU TIME   0.17 MINUTES WAIT TIME
```

```
BEGIN
   INTEGER I;
   SWITCH FANG := SARA,ISU;
ISU:
   BEGIN
      INTEGER I;
      SWITCH SCHWARTZMAN := BRUNO, FANG @(1)@;
      I := 3;
BRUNO:I := I - 1;
      IF I = 2 THEN GO TO SCHWARTZMAN@(1)@ ELSE
      IF I = 0 THEN GO TO FANG @(I + 1)@;
      GO TO FANG@(2)@
   END;
SARA:
   I := 0
END
```

```
ORIGIN=     1

3
C(ACTUAL(BODY=5,LEVEL=5,QUAL=,UNCERTYPE=LABEL))
C(ACTUAL(BODY=4,LEVEL=5,QUAL=,UNCERTYPE=LABEL))

8
C(ACTUAL(BODY=10,LEVEL=8,QUAL=,UNCERTYPE=LABEL))
C(ACTUAL(BODY=9,LEVEL=8,QUAL=,UNCERTYPE=LABEL))

9
VAL(ADDR=(LN=4,DA=2))
C(INTEGER(VALUE=1))
INX(VALUE)
RET

10
VAL(ADDR=(LN=5,DA=3))
RET

4
VAL(ADDR=(LN=4,DA=3))
RET

5
VAL(ADDR='LN=4,DA=4))
RET

7
```

```
C(INTEGER)
C(SWITCHLIST=0,LENGTH=2)
C(LABEL(SEGMENT=7,DISP=7))
ACR(ADDR=(LN=5,CN=1))
C(INTEGER(VALUE=3))
STG(FALSE)
ALR(ADDR=(LN=5,CN=1))
VAL(ADDR=(LN=5,CN=1))
C(INTEGER(VALUE=1))
AR(-)
STG(FALSE)
VAL(ADDR=(LN=5,CN=1))
C(INTEGER(VALUE=2))
COMP(=)
IFJ(21)
VAL(ADDR=(LN=5,CN=2))
C(INTEGER(VALUE=1))
INX(VALUE)
GO
GO(31)
VAL(ADDR=(LN=5,CN=1))
C(INTEGER(VALUE=0))
COMP(=)
IFJ(31)
VAL(ADDR=(LN=4,CN=2))
VAL(ADDR=(LN=5,CN=1))
C(INTEGER(VALUE=1))
AR(+)
INX(VALUE)
GO
VAL(ADDR=(LN=4,CN=2))
C(INTEGER(VALUE=2))
INX(VALUE)
GO
RET

1
C(INTEGER)
C(SWITCHLIST=3,LENGTH=2)
C(LABEL(SEGMENT=1,DISP=5))
C(LABEL(SEGMENT=1,DISP=7))
MARK
ENT(LEVEL=5,BODY=7)
ACR(ADDR=(LN=4,CN=1))
C(INTEGER(VALUE=0))
STG(FALSE)
RET


 RALPH JOB STATISTICS --  0.22 MINUTES EXECUTION TIME
                          0.06 MINUTES CPU TIME   0.16 MINUTES WAIT TIME
```

```
BEGIN  CLASS B
       (L1;  INTEGER L;  BEGIN
       CLASS F;  BEGIN
             BOOLEAN G;
             END;
       F CLASS I;
             ;
       REF (I) S;
       S :- NEW I;
       END;
B (5) BEGIN
      INSPECT  S DO O := L<O
      END
END
```



```
ORIGIN=     1

4
NEW OBJECT(BODY=3,IS=4,PREFIX=0)

7
NEW OBJECT(BODY=6,IS=7,PREFIX=0)

6
C(BOOLEAN)
DET(TER)

10
NEW OBJECT(BODY=9,IS=10,PREFIX=7)

9
C(BOOLEAN)
DET(TER)

3
CME(O=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
C(CLASS(PROTOTYPE=7,LEVEL=7))
C(CLASS(PROTOTYPE=10,LEVEL=7))
C(REF(QUAL=9,VALUE=))
ADR(ADDR=(LN=6,OA=4))
MARK
VAL(ADDR=(LN=6,OA=3))
GEN(O)
RET
STO(FALSE)
DET(TER)
```

```
12
NEW OBJECT(BODY=11,IS=12,PREFIX=4)

13
C(INTEGER(VALUE=5))
RET

1
C(CLASS(PROTOTYPE=4,LEVEL=5))
MARK
C(ACTUAL(BODY=13,LEVEL=5,QUAL=,VALUETYPE=INTEGER))
C(CLASS(PROTOTYPE=12,LEVEL=5))
GEN(1)
RET
DEL
RET

11
CHECK=1,DECLOS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
C(CLASS(PROTOTYPE=7,LEVEL=7))
C(CLASS(PROTOTYPE=10,LEVEL=7))
C(REF(QUAL=),VALUE=))
C(REF)
ADR(ADDR=(LN=6,ON=4))
MARK
VAL(ADDR=(LN=6,ON=3))
GEN(3)
RET
STO(FALSE)
ADR(ADDR=(LN=6,LN=5))
VAL(ADDR=(LN=6,ON=4))
STO(TRUE )
C(REF(QUAL=-1,VALUE=))
COMP(=/=)
IFJ(21)
MARK
ENT(LEVEL=7,BODY=16)
GO(21)
DET(TER)

16
VAL(ADDR=(LN=6,ON=5))
ADR(ADDR=(LN=8,ON=1),REM)
VAL(ADDR=(LN=6,ON=1))
C(INTEGER(VALUE=0))
COMP(<)
STO(FALSE)
RET
```

```
 RALPH JOB STATISTICS --  0.22 MINUTES EXECUTION TIME
                          0.06 MINUTES CPU TIME   0.17 MINUTES WAIT TIME
```

```
BEGIN
   CLASS A(I,J); VALUE I; INTEGER I,J;
   VIRTUAL: LABEL L; REAL PROCEDURE P;
   BEGIN
      REAL PROCEDURE P(I); INTEGER I; P := I;
      REAL K; GO TO L; INNER; L1: L: K := P(J)
   END;

   A CLASS B (M); REAL M;
   BEGIN
      REAL PROCEDURE P(I); NAME I; INTEGER I; P := 200*I;
      BOOLEAN BOO;
      L2: K := K - M; L: BOO := IF K<0 THEN K=M = 0 ELSE FALSE;
      IF K<M THEN GO TO L1 ELSE GO TO L2
   END;

   REF (B) X, Y;
   REAL K; BOOLEAN BOO;
   X :- NEW B(2,3,4); Y :- NEW B(3,4,5);
   K := X.P(3); BOO := X.BOO; X.K := Y.K := Y.P(K)
END
```

```
ENTERING GARBAGE COLLECTION

NUMBER OF CELLS COLLECTED=( 0238,  408,  145)


ORIGIN=     1

4
NEW OBJECT(BODY=3,IS=4,PREFIX=0)

19
C(INTEGER(VALUE=21)
RET

20
C(INTEGER(VALUE=3))
RET

21
C(INTEGER(VALUE=4))
RET
```

```
22
C(INTEGER(VALUE=3))
RET

23
C(INTEGER(VALUE=4))
RET

24
C(INTEGER(VALUE=5))
RET

12
CHE(ID=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=NAME,CLASS=FALSE)
ADR(ADDR=(LN=7,ON=1))
C(INTEGER(VALUE=2))
VAL(ADDR=(LN=7,ON=2))
AR(99)
STO(FALSE)
RET

25
C(INTEGER(VALUE=3))
RET

10
New OBJECT(BODY=9,IS=10,PREFIX=4)

5
CHE(ID=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=FALSE)
ADR(ADDR=(LN=7,ON=1))
VAL(ADDR=(LN=7,ON=2))
STO(FALSE)
RET

3
CHE(ID=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
CHE(ID=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
C(LABEL(SEGMENT=3,DISP=9))
C(PROCEDURE(LEVEL=7,SEGMENT=5))
C(REAL)
C(LABEL(SEGMENT=3,DISP=9))
VAL(ADDR=(LN=6,ON=3))
GO
ADR(ADDR=(LN=6,ON=5))
MARK
C(RET)
C(REAL)
C(ACTUAL(BODY=28,LEVEL=7,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=6,ON=4))
ENT
STO(FALSE)
DET(TER)

28
ADR(ADDR=(LN=6,ON=2))
RET

9
CHE(ID=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
```

```
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
C(LABEL(SEGMENT=9,DISP=17))
C(PROCEDURE(LEVEL=7,SEGMENT=12))
C(REAL)
C(LABEL(SEGMENT=9,DISP=39))
CHECK=3,GENUS=(KIND=SIMPLE,TYPE=REAL,QUAL=),MODE=VALUE,CLASS=TRUE )
C(BOOLEAN)
C(LABEL(SEGMENT=9,DISP=12))
VAL(ADDR=(LA=6,DA=3))
GO
ADR(ADDR=(LA=6,DA=5))
VAL(ADDR=(LA=6,DA=5))
VAL(ADDR=(LA=6,DA=7))
AR(-)
STO(FALSE)
ADR(ADDR=(LA=6,DA=8))
VAL(ADDR=(LA=6,DA=5))
C(INTEGER(VALUE=0))
COMP(<)
IFJ(28)
VAL(ADDR=(LA=6,DA=5))
VAL(ADDR=(LA=6,DA=7))
AR(+)
C(INTEGER(VALUE=0))
COMP(=)
GO(29)
C(BOOLEAN(VALUE="FALSE"))
STO(FALSE)
VAL(ADDR=(LA=6,DA=5))
VAL(ADDR=(LA=6,DA=7))
COMP(<)
IFJ(37)
VAL(ADDR=(LA=6,DA=6))
GO
GO(39)
VAL(ADDR=(LA=6,DA=9))
GO
ADR(ADDR=(LA=6,DA=5))
MARK
C(RET)
C(REAL)
C(ACTUAL(BODY=28,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LA=6,DA=4))
ENT
STO(FALSE)
DET(TER)

1
C(CLASS(PROTOTYPE=4,LEVEL=5))
C(CLASS(PROTOTYPE=10,LEVEL=5))
C(REF(QUAL=9,VALUE=))
C(REF(QUAL=9,VALUE=))
C(REAL)
C(BOOLEAN)
ADR(ADDR=(LA=4,DA=3))
MARK
C(ACTUAL(BODY=19,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
C(ACTUAL(BODY=20,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
C(ACTUAL(BODY=21,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LA=4,DA=2))
```

```
GEN(3)
RET
STO(FALSE)
ADR(ADDR=(LN=4,DN=4))
MARK
C(ACTUAL(BODY=22,LEVEL=5,QUAL=,UNCERTYPE=INTEGER))
C(ACTUAL(BODY=23,LEVEL=5,QUAL=,UNCERTYPE=INTEGER))
C(ACTUAL(BODY=24,LEVEL=5,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=4,DN=2))
GEN(3)
RET
STO(FALSE)
ADR(ADDR=(LN=4,DN=5))
MARK
C(RET)
C(REAL)
C(ACTUAL(BODY=25,LEVEL=5,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=4,DN=3))
VAL(ADDR=(LN=6,DN=4),REM)
ENT
STO(FALSE)
ADR(ADDR=(LN=4,DN=6))
VAL(ADDR=(LN=4,DN=3))
VAL(ADDR=(LN=6,DN=6),REM)
STO(FALSE)
VAL(ADDR=(LN=4,DN=3))
ADR(ADDR=(LN=6,DN=5),REM)
VAL(ADDR=(LN=4,DN=4))
ADR(ADDR=(LN=6,DN=5),REM)
MARK
C(RET)
C(REAL)
C(ACTUAL(BODY=32,LEVEL=5,QUAL=,UNCERTYPE=REAL))
VAL(ADDR=(LN=4,DN=4))
VAL(ADDR=(LN=6,DN=4),REM)
ENT
STO(TRUE )
STO(FALSE)
RET

$2
ADR(ADDR=(LN=4,DN=5))
RET


RALPH JOB STATISTICS --  0.30 MINUTES EXECUTION TIME
                         0.13 MINUTES CPU TIME   0.17 MINUTES WAIT TIME
```

```
COMMENT  PROGRAMS (VIIII) AND (IX) APPEAR IN DAML SIGAM 7108;

BEGIN
   CLASS HISTOGRAM(X,N); ARRAY X; INTEGER N;
   BEGIN INTEGER NM; INTEGER ARRAY T0[0:N]0;
      PROCEDURE TABULATE (Y); REAL Y;
      BEGIN INTEGER I;
         WHILE IF I<N THEN  Y<X0[I+1]0
                      ELSE  FALSE DO  I := I+1;
         T0[I]0 := T0[I+1]0; NM := NM+1
      END  TABULATE;
      REAL PROCEDURE FREQ (I); INTEGER I;
           FREQ := T0[I]0/NM;
   END HISTOGRAM;

   ARRAY Z0[1:10]0; REF(HISTOGRAM) HG;  REAL A;
   HG :- NEW HISTOGRAM(Z,10);
   HG.TABULATE(10); A := HG.FREQ(2)
END
```

```
ORIGIN=     1

4
NEW OBJECT(NODY=3,IS=4,PREFIX=0)

14
CI(INTEGER(VALUE=10))
RET

15
CI(INTEGER(VALUE=10))
RET

13
ACR(ADDR=(LN=4,DA=2))
RET

12
CHE(ID=2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=1,MODE=VALUE,CLASS=FALSE)
ADR(ADDR=(LN=7,DA=1))
VAL(ADDR=(LN=6,DA=4))
VAL(ADDR=(LN=7,DA=2))
INX(VALUE)
VAL(ADDR=(LN=6,DA=3))
AR(/)
STO(FALSE)
```

```
RET

3
CHECK(1,GENUS=(KIND=ARRAY,TYPE=REAL,QUAL=3),MODE=REFERENCE,CLASS=TRUE )
CHECK(2,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=3),MODE=VALUE,CLASS=TRUE )
C(INTEGER)
C(INTEGER(VALUE=0))
VAL(ADDR=(LN=0,CN=2))
MAK(GENUS=(TYPE=INTEGER    KIND=ARRAY),N=1,COPIES=0)
C(PROCEDURE(LEVEL=7,SEGMENT=6))
C(PROCEDURE(LEVEL=7,SEGMENT=12))
DET(TER)

6
CHECK(2,GENUS=(KIND=SIMPLE,TYPE=REAL,QUAL=),MODE=VALUE,CLASS=FALSE)
C(INTEGER)
VAL(ADDR=(LN=7,CN=3))
VAL(ADDR=(LN=6,CN=2))
COMP(<)
IFJ(15)
VAL(ADDR=(LN=7,CN=2))
VAL(ADDR=(LN=6,LN=1))
VAL(ADDR=(LN=7,CN=3))
C(INTEGER(VALUE=1))
AR(+)
INX(VALUE)
COMP(<)
GO(16)
C(BOOLEAN(VALUE="FALSE"))
IFJ(23)
ADR(ADDR=(LN=7,CN=3))
VAL(ADDR=(LN=7,CN=3))
C(INTEGER(VALUE=1))
AR(+)
STO(FALSE)
GO(3)
VAL(ADDR=(LN=6,CN=4))
VAL(ADDR=(LN=7,CN=3))
INX(ADDR)
VAL(ADDR=(LN=6,CN=4))
VAL(ADDR=(LN=7,CN=3))
C(INTEGER(VALUE=1))
AR(+)
INX(VALUE)
STO(FALSE)
ADR(ADDR=(LN=0,CN=3))
VAL(ADDR=(LN=6,CN=3))
C(INTEGER(VALUE=1))
AR(+)
STO(FALSE)
RET

16
C(INTEGER(VALUE=2))
RET

1
C(CLASS(PROTOTYPE=4,LEVEL=5))
C(INTEGER(VALUE=1))
C(INTEGER(VALUE=10))
```

```
MAK(GENLS=(TYPE=REAL    KIND=ARRAY),N=1,COPIES=0)
C(REF(QUAL=3,VALUE=1))
C(REAL)
ADR(ADDR=(LN=4,CA=3))
MARK
C(ACTUAL(BODY=13,LEVEL=5,QUAL=,UNDERTYPE=REAL))
C(ACTUAL(BODY=14,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=4,CA=1))
GEN(2)
RET
STC(FALSE)
MARK
C(RET)
C(INTEGER)
C(ACTUAL(BODY=15,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=4,CA=3))
VAL(ADDR=(LN=6,CA=5),REM)
ENT
DEL
ALN(ADDR=(LN=4,CA=4))
MARK
C(RET)
C(REAL)
C(ACTUAL(BODY=16,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=4,CA=3))
VAL(ADDR=(LN=6,CA=6),REM)
ENT
STC(FALSE)
RET


 RALPH JOB STATISTICS --  0.24 MINUTES EXECUTION TIME
                          0.09 MINUTES CPU TIME   0.15 MINUTES WAIT TIME
```

```
BEGIN
   CLASS TRANSPERM (N); INTEGER N;
   BEGIN INTEGER ARRAY P#(1:N)#; INTEGER Q; BOOLEAN MORE;

         CLASS COMPONENT (K); INTEGER K;
         BEGIN P#(K)# := K; DETACH END;
         COMPONENT CLASS TRANSPOSITION (NEXT);
               REF (COMPONENT) NEXT;
               VIRTUAL: PROCEDURE TRANSPOSE;
         BEGIN INTEGER I;
               WHILE TRUE DO
               BEGIN  I := 1 ; WHILE I <= K-1  DO
                      BEGIN TRANSPOSE; DETACH; I := I+1 END;
                      I := K-1; TRANSPOSE; DETACH;
                      RESUME(NEXT); DETACH
               END
         END;

         TRANSPOSITION CLASS EVEN;
         BEGIN PROCEDURE TRANSPOSE;
               BEGIN Q := P#(K)#; P#(K)# := P#(K+1)#; P#(K+1)# := Q END;
         END;

         TRANSPOSITION CLASS ODD;
         BEGIN PROCEDURE TRANSPOSE;
               BEGIN Q := P#(I)#; P#(I)# := P#(K+1)#; P#(K+1)# := Q END;
         END;

         COMPONENT CLASS FINAL; MORE := FALSE;
         REF (COMPONENT) PM;
         PM :- NEW FINAL (N);
         Q := N-1; WHILE Q>=2 DO
         BEGIN PM :- IF Q DIV 2*2 = Q THEN NEW EVEN(Q,PM)
                                      ELSE NEW ODD (Q,PM);
               Q := Q-1 END;
         PM :- NEW EVEN(1,PM); MORE := TRUE
   END TRANSPERM;

   INSPECT NEW TRANSPERM(5) DO
   WHILE MORE DO RESUME(PM)
END
```

```
ENTERING GARBAGE COLLECTION
```

```
NUMBER OF CELLS COLLECTED=( 3458,  560,  475)


ENTERING GARBAGE COLLECTION

NUMBER OF CELLS COLLECTED=( 2466,  570,  489)


ENTERING GARBAGE COLLECTION

NUMBER OF CELLS COLLECTED=( 5425,  653,  374)


ORIGIN=    1

4
NEW OBJECT(BODY=3,IS=4,PREFIX=0)

7
NEW OBJECT(BODY=6,IS=7,PREFIX=0)

34
CI(INTEGER(VALUE=1))
RET

39
CI(INTEGER(VALUE=5))
RET

38
VAL(ADDR=(LN=4,CN=2))
VAL(ADDR=(LN=8,DA=4),REM)
IFJ(8)
VAL(ADDR=(LN=4,CN=2))
AGR(ADDR=(LN=6,CN=10),REM)
RES
GO(1)
RET

24
NEW OBJECT(BODY=23,IS=24,PREFIX=7)

20
NEW OBJECT(BODY=19,IS=20,PREFIX=9)

16
NEW OBJECT(BODY=15,IS=16,PREFIX=9)

9
NEW OBJECT(BODY=8,IS=9,PREFIX=7)

6
CHE(ID=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
VAL(ADDR=(LN=6,CN=2))
VAL(ADDR=(LN=8,DA=1))
INX(ADDR)
VAL(ADDR=(LN=8,CN=1))
```

```
STO(FALSE)
RET
DEF(TER)

1
CALLASS(PROTOTYPE=4,LEVEL=5))
CREF)
ADR(ADDR=(LN=4,CA=2))
MARK
CALACTUAL(BODY=39,LEVEL=5,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=4,CA=1))
GEN(1)
RET
STO(TRUE )
CREF(QUAL=-1,VALUE=))
COMP(=/=)
IFJ(16)
MARK
EAT(LEVEL=5,BODY=39)
GO(16)
RET

22
ADR(ADDR=(LN=6,CA=3))
VAL(ADDR=(LN=8,CA=2))
VAL(ADDR=(LN=6,CA=1))
INX(VALUE)
STO(FALSE)
VAL(ADDR=(LN=6,CA=2))
VAL(ADDR=(LN=8,CA=1))
INX(ADDR)
VAL(ADDR=(LN=6,CA=2))
VAL(ADDR=(LN=8,CA=1))
C(INTEGER(VALUE=1))
AR(+)
INX(VALUE)
STO(FALSE)
VAL(ADDR=(LN=6,CA=2))
VAL(ADDR=(LN=8,CA=1))
C(INTEGER(VALUE=1))
AR(+)
INX(ADDR)
VAL(ADDR=(LN=6,CA=3))
STO(FALSE)
RET

18
ADR(ADDR=(LN=6,CA=3))
VAL(ADDR=(LN=8,CA=2))
VAL(ADDR=(LN=6,CA=1))
INX(VALUE)
STO(FALSE)
VAL(ADDR=(LN=6,CA=2))
VAL(ADDR=(LN=8,CA=1))
INX(ADDR)
VAL(ADDR=(LN=6,CA=2))
VAL(ADDR=(LN=8,CA=1))
C(INTEGER(VALUE=1))
AR(+)
INX(VALUE)
```

```
STO(FALSE)
VAL(ADDR=(LN=6,CN=2))
VAL(ADDR=(LN=8,CN=1))
C(INTEGER(VALUE=1))
AR(+)
INX(ADDR)
VAL(ADDR=(LN=6,CN=3))
STO(FALSE)
RET

8
CHEIO=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
CHEIO=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
C(PROCEDURE)
C(INTEGER)
VAL(ADDR=(LN=6,CN=2))
VAL(ADDR=(LN=8,CN=1))
INX(ADDR)
VAL(ADDR=(LN=8,CN=1))
STO(FALSE)
RET
C(BOOLEAN(VALUE="TRUE"))
EFJ(51)
ADR(ADDR=(LN=8,CN=4))
C(INTEGER(VALUE=1))
STO(FALSE)
VAL(ADDR=(LN=8,CN=4))
VAL(ADDR=(LN=8,CN=1))
C(INTEGER(VALUE=1))
AR(-)
CCMP(<=)
EFJ(35)
MARK
C(RET)
C(INTEGER:
VAL(ADDR=(LN=8,CN=3))
ENT
DEL
DET
ADR(ADDR=(LN=8,CN=4))
VAL(ADDR=(LN=8,CN=4))
C(INTEGER(VALUE=1))
AR(+)
STO(FALSE)
GO(16)
ADR(ADDR=(LN=8,CN=4))
VAL(ADDR=(LN=8,CN=1))
C(INTEGER(VALUE=1))
AR(-)
STO(FALSE)
MARK
C(RET)
C(INTEGER)
VAL(ADDR=(LN=8,CN=3))
ENT
DEL
DET
ADR(ADDR=(LN=8,CN=2))
RES
DET
```

```
GO(11)
DET(TER)

15
CHE(ID=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
CHE(ID=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
C(PROCEDURE(LEVEL=4,SEGMENT=18))
C(INTEGER)
VAL(ADDR=(LN=6,DN=2))
VAL(ADDR=(LN=8,DN=1))
INX(ADDR)
VAL(ADDR=(LN=8,LN=1))
STO(FALSE)
DET
C(BOOLEAN(VALUE="TRUE"))
IFJ(51)
ADR(ADDR=(LN=8,DN=4))
C(INTEGER(VALUE=1))
STO(FALSE)
VAL(ADDR=(LN=8,DN=4))
VAL(ADDR=(LN=8,DN=1))
C(INTEGER(VALUE=1))
BR(-)
COMP(<=)
IFJ(35)
MARK
C(REF)
C(INTEGER)
VAL(ADDR=(LN=8,DN=3))
ENT
DEL
DET
ADR(ADDR=(LN=8,DN=4))
VAL(ADDR=(LN=8,DN=4))
C(INTEGER(VALUE=1))
BR(+)
STO(FALSE)
GO(16)
ADR(ADDR=(LN=8,DN=4))
VAL(ADDR=(LN=8,DN=1))
C(INTEGER(VALUE=1))
BR(-)
STO(FALSE)
MARK
C(REF)
C(INTEGER)
VAL(ADDR=(LN=8,DN=3))
ENT
DEL
DET
ADR(ADDR=(LN=8,DN=2))
RES
DET
GO(11)
DET(TER)

19
CHE(ID=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
CHE(ID=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
C(PROCEDURE(LEVEL=9,SEGMENT=22))
```

```
C(INTEGER)
VAL(ADDR=(LN=6,ON=2))
VAL(ADDR=(LA=8,CA=1))
INX(ADDR)
VAL(ADDR=(LA=8,CN=1))
STO(FALSE)
DET
C(BOOLEAN(VALUE="TRUE"))
EFJ(51)
ACR(ADDR=(LA=4,CA=4))
C(INTEGER(VALUE=1))
STO(FALSE)
VAL(ADDR=(LN=8,DA=4))
VAL(ADDR=(LA=8,DA=1))
C(INTEGER(VALUE=1))
AR(-)
COMP(<=)
EFJ(35)
MARK
C(RET)
C(INTEGER)
VAL(ADDR=(LN=8,CA=3))
ENT
DEL
DET
ACR(ADDR=(LA=4,CA=4))
VAL(ADDR=(LA=8,DA=4))
C(INTEGER(VALUE=1))
AR(+)
STO(FALSE)
GO(16)
ACR(ADDR=(LA=4,CA=4))
VAL(ADDR=(LA=8,CA=1))
C(INTEGER(VALUE=1))
AR(-)
STO(FALSE)
MARK
C(RET)
C(INTEGER)
VAL(ADDR=(LN=8,ON=3))
ENT
DEL
DET
ADR(ADDR=(LA=8,CA=2))
RES
DET
GO(11)
DET(TER)

23
CHEIC=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
VAL(ADDR=(LN=6,CA=2))
VAL(ADDR=(LA=8,CA=1))
INX(ADDR)
VAL(ADDR=(LN=8,CN=1))
STO(FALSE)
DET
ACR(ADDR=(LA=6,CA=4))
C(BOOLEAN(VALUE="FALSE"))
STO(FALSE)
```

```
DEFINER

44
ADR(ADDR=(LN=6,CA=1))
RET

47
ADR(ADDR=(LN=6,CA=3))
RET

50
ADR(ADDR=(LN=6,CA=10))
RET

53
ADR(ADDR=(LN=6,CA=3))
RET

56
ADR(ADDR=(LN=6,CA=16))
RET

59
ADR(ADDR=(LN=6,CA=16))
RET

3
CPE(ID=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
C(INTEGER(VALUE=1))
VAL(ADDR=(LN=6,CA=1))
MAK(GENUS=(TYPE=INTEGER    KIND=ARRAY),N=1,COPIES=0)
C(INTEGER)
C(BOOLEAN)
C(CLASS(PROTOTYPE=7,LEVEL=7))
C(CLASS(PROTOTYPE=9,LEVEL=7))
C(CLASS(PROTOTYPE=16,LEVEL=7))
C(CLASS(PROTOTYPE=20,LEVEL=7))
C(CLASS(PROTOTYPE=24,LEVEL=7))
C(REF(QUAL=6,VALUE=3))
ADR(ADDR=(LN=6,CA=10))
MARK
C(ACTUAL(BODY=44,LEVEL=7,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=6,CA=9))
GEN(1)
RET
STC(FALSE)
ADR(ADDR=(LN=6,CA=3))
VAL(ADDR=(LN=6,CA=1))
C(INTEGER(VALUE=1))
AR(-)
STC(FALSE)
VAL(ADDR=(LN=6,CA=3))
C(INTEGER(VALUE=2))
COMP(>=)
IFJ(SE)
ADR(ADDR=(LN=6,CA=10))
VAL(ADDR=(LN=6,CA=3))
C(INTEGER(VALUE=2))
AR(DIV)
C(INTEGER(VALUE=2))
```

Reproduced from best available copy.

```
AR(*)
VAL(ADDR=(LN=6,CN=3))
COMP(=)
EFJ(45)
MARK
C(ACTUAL(BODY=47,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
C(ACTUAL(BODY=50,LEVEL=7,QUAL=6,UNDERTYPE=REF))
VAL(ADDR=(LN=6,CN=7))
GEN(2)
RET
GO(51)
MARK
C(ACTUAL(BODY=53,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
C(ACTUAL(BODY=56,LEVEL=7,QUAL=6,UNDERTYPE=REF))
VAL(ADDR=(LN=6,CN=8))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=6,CN=3))
VAL(ADDR=(LN=6,CN=3))
C(INTEGER(VALUE=1))
AR(-)
STO(FALSE)
GO(25)
ADR(ADDR=(LN=6,CN=10))
MARK
C(ACTUAL(BODY=34,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
C(ACTUAL(BODY=59,LEVEL=7,QUAL=6,UNDERTYPE=REF))
VAL(ADDR=(LN=6,CN=7))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=6,CN=4))
C(BOOLEAN(VALUE="TRUE"))
STO(FALSE)
DET(TER)
```

```
 RALPH JOB STATISTICS --  0.45 MINUTES EXECUTION TIME
                          0.25 MINUTES CPU TIME   0.20 MINUTES WAIT TIME
```

```
COMMENT  THIS PROGRAM APPEARS IN ICHBIAH & MORSE BEIGH 7210;

BEGIN
CLASS SYMBOL;

BEGIN
   CLASS EXPR; VIRTUAL:REF(E.PR) PROCEDURE DERIV;
   BEGIN
      REF(EXPR)DELTA;
   END EXPR;

   EXPR CLASS CONSTANT(K); REAL K;
        BEGIN
           REF(EXPR)PROCEDURE DERIV(X); REF(VARIABLE)X;
                DERIV :- DELTA :- ZERO;
        END CONSTANT;

   EXPR CLASS VARIABLE(ID); INTEGER ID;
        BEGIN
           REF(EXPR)PROCEDURE DERIV(X); REF(VARIABLE)X;
           IF X == THIS VARIABLE THEN
             DERIV :- DELTA :- ONE
           ELSE
             DERIV :- DELTA :- ZERO;
        END VARIABLE;

   EXPR CLASS PAIR(LEFT, RIGHT); REF(EXPR) LEFT,RIGHT;;

   PAIR CLASS SUM;
        BEGIN
           REF(EXPR)PROCEDURE DERIV(X); REF(VARIABLE)X;
           BEGIN
              REF(EXPR)LPRIME,RPRIME;
              LPRIME :- LEFT.DERIV(X);
              RPRIME :- RIGHT.DERIV(X);
              DELTA :- IF LPRIME == ZERO THEN
                         RPRIME
                       ELSE
                         IF RPRIME == ZERO THEN
                           LPRIME
                         ELSE
                           NEW SUM(LPRIME,RPRIME);
              DERIV :- DELTA
           END DERIV;
        END SUM;

   PAIR CLASS DIFF;
        BEGIN
           REF(EXPR)PROCEDURE DERIV(X); REF(VARIABLE)X;
           BEGIN
              REF(EXPR)LPRIME,RPRIME;
              LPRIME :- LEFT.DERIV(X);
              RPRIME :- RIGHT.DERIV(X);
```

```
            DELTA :- IF APRIME == ZERO THEN
                       LPRIME
                     ELSE
                       NEW DIFF(LPRIME,APRIME);
            DERIV :- DELTA
          END OF IF;
        END DIFF;

        REF(CONSTANT)ZERO,ONE;

        COMMENT THE INITIAL OPERATIONS OF THE CLASS SYMBOL
        ARE THE FOLLOWING;
        ZERO :- NEW CONSTANT(0);
        ONE :- NEW CONSTANT(1);
        ZERO.DELTA :- ONE.DELTA :- ZERO;
END SYMBOL;

SYMBOL BEGIN
          REF(VARIABLE)X,Y,Z;
          REF(EXPR)U,V,E,F;
          X :- NEW VARIABLE(1); Y :- NEW VARIABLE(2);
          U :- NEW SUM(X,Y);
          Z :- NEW VARIABLE(3);
          V :- NEW DIFF(Z,NEW CONSTANT(4));
          E :- NEW DIFF(U,V);
          F :- E.DERIV(X)
        END
END
```

```
ENTERING GARBAGE COLLECTION
NUMBER OF CELLS COLLECTED=( 3247,  466,  523)

ENTERING GARBAGE COLLECTION
NUMBER OF CELLS COLLECTED=( 3091,  502,  458)

ENTERING GARBAGE COLLECTION
NUMBER OF CELLS COLLECTED=( 5120,  568,  425)

ENTERING GARBAGE COLLECTION
NUMBER OF CELLS COLLECTED=( 5625,  658,  696)
```

```
ORIGIN=     1

4
NEW OBJECT(BODY=3,IS=4,PREFIX=0)

7
NEW OBJECT(BODY=6,IS=7,PREFIX=0)

6
C(PROCEDURE)
C(REF(QUAL=6,VALUE=))
DET(TER)

45
C(INTEGER(VALUE=0))
RET

46
C(INTEGER(VALUE=1))
RET

35
NEW OBJECT(BODY=34,IS=35,PREFIX=20)

22
NEW OBJECT(BODY=21,IS=22,PREFIX=20)

20
NEW OBJECT(BODY=19,IS=20,PREFIX=7)

19
C(PROCEDURE)
C(REF(QUAL=6,VALUE=))
CHECK=1,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
DET(TER)

21
C(PROCEDURE(LEVEL=9,SEGMENT=24))
C(REF(QUAL=6,VALUE=))
CHECK=1,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
DET(TER)

34
C(PROCEDURE(LEVEL=9,SEGMENT=37))
C(REF(QUAL=6,VALUE=))
CHECK=1,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=6),MODE=REFERENCE,CLASS=TRUE )
DET(TER)

14
NEW OBJECT(BODY=13,IS=14,PREFIX=7)

13
C(PROCEDURE(LEVEL=9,SEGMENT=16))
C(REF(QUAL=6,VALUE=))
CHECK=1,GENUS=(KIND=SIMPLE,TYPE=INTEGER,QUAL=),MODE=VALUE,CLASS=TRUE )
```

```
DET(ITER)

10
NEW OBJECT(BODY=9,IS=10,PREFIX=7)

9
C(PROCEDURE(LEVEL=9,SEGMENT=12))
C(REF(QUAL=6,VALUE=1))
CHECK=1,GENUS=(KIND=SIMPLE,TYPE=REAL,QUAL=3,MODE=VALUE,CLASS=TRUE )
DET(ITER)

12
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=13),MODE=REFERENCE,CLASS=FALSE)
ADR(ADDR=(LN=9,CA=1))
ADR(ADDR=(LN=8,CA=2))
VAL(ADDR=(LN=6,CA=7))
STO(TRUE )
STO(FALSE)
RET

16
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=13),MODE=REFERENCE,CLASS=FALSE)
VAL(ADDR=(LN=9,CA=2))
C(REF(QUAL=13,VALUE=THIS))
COMP(==)
IFJ(12)
ADR(ADDR=(LN=9,CA=1))
ADR(ADDR=(LN=8,CA=2))
VAL(ADDR=(LN=6,CA=6))
STO(TRUE )
STO(FALSE)
GO(17)
ADR(ADDR=(LN=9,CA=1))
ADR(ADDR=(LN=8,CA=2))
VAL(ADDR=(LN=6,CA=7))
STO(TRUE )
STO(FALSE)
RET

51
ADR(ADDR=(LN=9,CA=2))
RET

54
ADR(ADDR=(LN=9,CA=2))
RET

57
ADR(ADDR=(LN=9,CA=3))
RET

37
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=13),MODE=REFERENCE,CLASS=FALSE)
C(REF(QUAL=6,VALUE=1))
C(REF(QUAL=6,VALUE=1))
ADR(ADDR=(LN=9,CA=3))
MARK
C(RET)
C(REF)
C(ACTUAL(BODY=51,LEVEL=10,QUAL=13,UNDERTYPE=REF))
```

Reproduced from best available copy.

```
VAL(ADDR=(LN=8,OA=3))
VAL(ADDR=(LN=8,OA=1),REM)
ENT
STO(FALSE)
ADR(ADDR=(LN=9,OA=4))
MARK
C(RET)
C(REF)
C(ACTUAL(BODY=54,LEVEL=10,QUAL=13,UNDERTYPE=REF))
VAL(ADDR=(LN=8,OA=4))
VAL(ADDR=(LN=8,OA=1),REM)
ENT
STO(FALSE)
ADR(ADDR=(LN=8,OA=2))
VAL(ADDR=(LN=9,OA=4))
VAL(ADDR=(LN=8,OA=7))
COMP(==)
BFJ(29)
VAL(ADDR=(LN=9,OA=3))
GOTO(35)
MARK
C(ACTUAL(BODY=57,LEVEL=10,QUAL=6,UNDERTYPE=REF))
C(ACTUAL(BODY=60,LEVEL=10,QUAL=6,UNDERTYPE=REF))
VAL(ADDR=(LN=6,OA=6))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=9,OA=1))
VAL(ADDR=(LN=8,OA=2))
STO(FALSE)
RET

60
ADR(ADDR=(LN=9,OA=4))
RET

63
ADR(ADDR=(LN=9,OA=2))
RET

66
ADR(ADDR=(LN=9,OA=2))
RET

69
ADR(ADDR=(LN=9,OA=3))
RET

24
CHECK=2,GENUS=(KIND=SIMPLE,TYPE=REF,QUAL=13),MODE=REFERENCE,CLASS=FALSE)
C(REF(QUAL=6,VALUE=3))
C(REF(QUAL=6,VALUE=3))
ADR(ADDR=(LN=9,OA=3))
MARK
C(RET)
C(REF)
C(ACTUAL(BODY=63,LEVEL=10,QUAL=13,UNDERTYPE=REF))
VAL(ADDR=(LN=8,OA=3))
VAL(ADDR=(LN=8,OA=1),REM)
ENT
```

```
STO(FALSE)
ADR(ADDR=(LN=9,CN=4))
MARK
C(RET)
C(REF)
C(ACTUAL(BODY=66,LEVEL=10,QUAL=13,UNDERTYPE=REF))
VAL(ADDR=(LN=8,CN=4))
VAL(ADDR=(LN=8,CN=1),REM)
ENT
STO(FALSE)
ADR(ADDR=(LN=8,CN=2))
VAL(ADDR=(LN=9,CN=3))
VAL(ADDR=(LN=8,CN=7))
COMP(==)
IFJ(29)
VAL(ADDR=(LN=9,CN=4))
GO(41)
VAL(ADDR=(LN=9,CN=4))
VAL(ADDR=(LN=8,CN=7))
COMP(==)
IFJ(35)
VAL(ADDR=(LN=9,CN=3))
GO(41)
MARK
C(ACTUAL(BODY=69,LEVEL=10,QUAL=6,UNDERTYPE=REF))
C(ACTUAL(BODY=72,LEVEL=10,QUAL=6,UNDERTYPE=REF))
VAL(ADDR=(LN=8,CN=5))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=9,CN=1))
VAL(ADDR=(LN=8,CN=2))
STO(FALSE)
RET

72
ADR(ADDR=(LN=9,CN=4))
RET

3
C(CLASS(PROTOTYPE=7,LEVEL=7))
C(CLASS(PROTOTYPE=10,LEVEL=7))
C(CLASS(PROTOTYPE=14,LEVEL=7))
C(CLASS(PROTOTYPE=20,LEVEL=7))
C(CLASS(PROTOTYPE=22,LEVEL=7))
C(CLASS(PROTOTYPE=35,LEVEL=7))
C(REF(QUAL=9,VALUE=))
C(REF(QUAL=9,VALUE=))
ADR(ADDR=(LN=8,CN=7))
MARK
C(ACTUAL(BODY=45,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=8,CN=2))
GEN(1)
RET
STO(FALSE)
ADR(ADDR=(LN=8,CN=8))
MARK
C(ACTUAL(BODY=46,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=8,CN=2))
GEN(1)
```

```
RET
STO(FALSE)
VAL(ADDR=(LN=6,CA=7))
ADR(ADDR=(LN=6,CA=2),REM)
VAL(ADDR=(LN=6,CA=8))
ADR(ADDR=(LA=8,CA=2),REM)
VAL(ADDR=(LN=6,CA=7))
STO(TRUE )
STO(FALSE)
DET(TER)

48
NEW OBJECT(BODY=47,IS=48,PREFIX=4)

1
C(CLASS(PROTOTYPE=4,LEVEL=5))
MARK
C(CLASS(PROTOTYPE=48,LEVEL=5))
GEN(0)
RET
CEL
RET

74
C(INTEGER(VALUE=1))
RET

75
C(INTEGER(VALUE=2))
RET

76
C(INTEGER(VALUE=3))
RET

81
C(INTEGER(VALUE=4))
RET

80
MARK
C(ACTUAL(BODY=81,LEVEL=8,QUAL=,UNDERTYPE=INTEGER))
ADR(ADDR=(LN=6,CA=2))
GEN(1)
RET
RET

86
ADR(ADDR=(LN=6,CA=9))
RET

89
ADR(ADDR=(LA=6,CA=10))
RET

92
ADR(ADDR=(LA=6,CA=1))
RET

95
```

Reproduced from best available copy.

```
ACR(ADDR=(LN=6,CN=12))
RET

98
ACR(ADDR=(LN=6,CN=13))
RET

47
C(CLASS(PROTOTYPE=7,LEVEL=7))
C(CLASS(PROTOTYPE=10,LEVEL=7))
C(CLASS(PROTOTYPE=14,LEVEL=7))
C(CLASS(PROTOTYPE=20,LEVEL=7))
C(CLASS(PROTOTYPE=22,LEVEL=7))
C(CLASS(PROTOTYPE=35,LEVEL=7))
C(REF(QUAL=5,VALUE=1))
C(REF(QUAL=9,VALUE=1))
C(REF(QUAL=13,VALUE=1))
C(REF(QUAL=13,VALUE=1))
C(REF(QUAL=13,VALUE=1))
C(REF(QUAL=6,VALUE=1))
C(REF(QUAL=6,VALUE=1))
C(REF(QUAL=6,VALUE=1))
C(REF(QUAL=6,VALUE=1))
ACR(ADDR=(LN=6,CN=7))
MARK
C(ACTUAL(BODY=45,LEVEL=7,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=6,CN=2))
GEN(1)
RET
STO(FALSE)
ADR(ADDR=(LN=6,CN=8))
MARK
C(ACTUAL(BODY=46,LEVEL=7,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=6,CN=2))
GEN(1)
RET
STO(FALSE)
VAL(ADDR=(LN=6,CN=7))
ADR(ADDR=(LN=6,CN=2),REM)
VAL(ADDR=(LN=6,CN=8))
ADR(ADDR=(LN=6,CN=2),REM)
VAL(ADDR=(LN=6,CN=7))
STO(TRUE )
STO(FALSE)
ACR(ADDR=(LN=6,CN=9))
MARK
C(ACTUAL(BODY=74,LEVEL=7,QUAL=,UNCERTYPE=INTEGER))
VAL(ADDR=(LN=6,LN=3))
GEN(1)
RET
STO(FALSE)
ACR(ADDR=(LN=6,CN=10))
MARK
C(ACTUAL(BODY=75,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=6,CN=3))
GEN(1)
RET
STO(FALSE)
ACR(ADDR=(LN=6,CN=12))
MARK
```

```
.
C(ACTUAL(BODY=86,LEVEL=7,QUAL=13,UNDERTYPE=REF))
C(ACTUAL(BODY=65,LEVEL=7,QUAL=13,UNDERTYPE=REF))
VAL(ADDR=(LN=6,DA=5))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=6,DA=11))
MARK
C(ACTUAL(BODY=78,LEVEL=7,QUAL=,UNDERTYPE=INTEGER))
VAL(ADDR=(LN=6,DA=3))
GEN(1)
RET
STO(FALSE)
ADR(ADDR=(LN=6,DA=13))
MARK
C(ACTUAL(BODY=92,LEVEL=7,QUAL=13,UNDERTYPE=REF))
C(ACTUAL(BODY=80,LEVEL=7,QUAL=5,UNDERTYPE=REF))
VAL(ADDR=(LN=6,DA=6))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=6,DA=14))
MARK
C(ACTUAL(BODY=95,LEVEL=7,QUAL=6,UNDERTYPE=REF))
C(ACTUAL(BODY=98,LEVEL=7,QUAL=6,UNDERTYPE=REF))
VAL(ADDR=(LN=6,DA=6))
GEN(2)
RET
STO(FALSE)
ADR(ADDR=(LN=6,DA=15))
MARK
C(RET)
C(REF)
C(ACTUAL(BODY=102,LEVEL=7,QUAL=13,UNDERTYPE=REF))
VAL(ADDR=(LN=6,DA=14))
VAL(ADDR=(LN=8,DA=1),REM)
ENT
STO(FALSE)
DET(TER)

102
ADR(ADDR=(LN=6,DA=9))
RET


  RALPH JOB STATISTICS --  0.44 MINUTES EXECUTION TIME
                           0.28 MINUTES CPU TIME   0.17 MINUTES WAIT TIME
```

Reproduced from best available copy.