The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation.

TECH MEMO

SDC

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 3417/200/00

AUTHOR Donna Firth Donna Firth

Jeff Barnett for P.A.

P. Abrahams

TECHNICAL Jeff Barnett

J. Barnett

RELEASE Clark W Weissman

C. Weissman, S.D.D.

Dean Anschultz

D. Anschultz, I.I.I.

for J. I. Schwartz

DATE 4/26/67 PAGE 1 OF 48 PAGES

(Page 2 is blank)

LISP 2 Language Specifications

ABSTRACT

This document describes the proposed syntax and semantics for the LISP 2 Source Language (SL) and Intermediate Language (IL) to be implemented on the IBM S/360 computer. The syntax of tokens is also included.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

1. INTRODUCTION

The Source Language (SL) and Intermediate Language (IL) of the LISP 2 system proposed for implementation on the IBM S/360 are described in this document. The LISP 2 system for the S/360 is an extension of the LISP 2 system currently operating on the Q-32 computer. Included in this document are descriptions of the syntax and semantics of data, tokens, types, expressions, variables, blocks, and declaratives. Specifications for other portions of the LISP 2 system for the IBM S/360 are included in other documents in this series (TM-3417).

2. TYPES, DATA, AND TOKENS

2.1 TERMINOLOGY

A field is a container or box capable of holding a representation of information. The contents of a field are known as a setting. One possible kind of setting is a locator, which is a rule for finding either a single field or a collection of fields.

A datum is the external representation of a computational object to be processed by a LISP 2 program. A data structure is the internal representation of a computational object to be processed by a LISP 2 program. A datum is in the form of a sequence of characters; a data structure is in the form of a collection of fields and settings. One of these settings is the value of the data structure. If the data structure consists of a single setting, then the value is that setting; otherwise the value is a locator from which the remaining parts of the data structure may be found. In either case, the value is a single setting.

Implicitly associated with every value is a type, which is a rule for determining the computational object represented by the value. Two values are said to be identical if and only if they have both the same setting and the same type. The operation of reading a datum results in obtaining a value whose associated data structure represents the same computational object as the datum; the operation of printing a value results in obtaining a datum that represents the same computational object as the data structure associated with the value.

2.2 TYPE INFORMATION

2.2.1 Syntax of Types

SL
IL

type ≡ simple-type|array-type|functional-type|n-tuple-type

simple-type ≡ GENERAL|BOOLEAN|REAL|INTEGER|BITS|FUNCTIONAL

SL

array-type ≡ [simple-type] ARRAY [dimensionality]

functional-type ≡ FUNCTIONAL (result-descriptor {, parameter-descriptor}* [, indef-parameter-descriptor])

parameter-descriptor ≡ {[variable-reference-mode]||type||[coordinate]}

result-descriptor ≡ parameter-descriptor|NOVALUE|{UNFIELDED||type}

indef-parameter-descriptor ≡ {[variable-reference-mode]||type||INDEF}

coordinate ≡ (unsigned-integer unsigned-integer)

IL

array-type ≡ (ARRAY [simple-type] {([integer] integer)}*)

functional-type ≡ (FUNCTIONAL result-descriptor {parameter-descriptor}* [indef-parameter-descriptor])

parameter-descriptor ≡ ({[variable-reference-mode]||type||[coordinate]})

result-descriptor ≡ parameter-descriptor|NOVALUE|{UNFIELDED||type}

indef-parameter-descriptor ≡ ({[variable-reference-mode] ||simple-type||INDEF})

coordinate ≡ (unsigned-integer . unsigned-integer)

SL
IL

variable-reference-mode ≡ DIRECT|INDIRECT

n-tuple-type ≡ identifier

2.2.2 Data Types

The various data structures in LISP 2 and their types are given in Table 1.

Arrays and n-tuples must have the proper number of elements; each element must be convertible to the expected type for the array or n-tuple component. All data is composed of one or more tokens. The following transformations may be used for data, SL, and IL.

OT space token ⟹ space space token

OT token space ⟹ token space space

Table 1. Data Types

Datum	Type
Number:	
Real	REAL
Integer	INTEGER
Octal	BITS
Unsigned-real	REAL
Unsigned-integer	INTEGER
Unsigned-octal	BITS
Boolean	BOOLEAN
String	GENERAL
Symbol	SYMBOL
Function-specifier	FUNCTIONAL
Nil	GENERAL
Node	NTUPLE
N-tuple	NTUPLE
Array	ARRAY

2.3 DATA SYNTAX

SL
IL

datum ≡ nonsymbol-element|symbol|nil|node

nonsymbol-element ≡ number|boolean|string|n-tuple|array
|function-specifier

node ≡ (datum$_{*+1}$ [. datum])

nil ≡ ()|NIL

symbol ≡ identifier|character-datum|special-spelling|mark-operator array ≡ ⌊array-typer dimensionality array-element$_{*+1}$⌋

array-element ≡ datum ⌊array-element$_{*+1}$⌋

dimensionality ≡ ⌊{[integer:] integer}$^{,}_{*+1}$⌋

function-specifier ≡ ⌊FUNCTION compound-var-name⌋

n-tuple ≡ ⌊n-tuple-name datum$_{*+1}$⌋

n-tuple-name ≡ identifier

2.4 TYPE CONVERSION

Under certain conditions, a value x of type α may be given when a value of a different type β is expected. In this case, there may be a value y of type β that can be used in place of x. This value y is then called the <u>homomorph of x in β</u>. A homomorph of a value will usually (but not always) represent the same computational object as the original value. A given value of type α may or may not have a homomorph in a different type β. A value is converted from type α to type β by finding the homomorph of the given value in β, if it exists. If it does not exist, the conversion is illegal. (See Table 2.)

β is said to be a <u>subtype</u> of α if every value in β can be converted to α and if the conversion does not change the actual setting. The subtype relationships among the different types are shown in the lattice of Figure 1. Here the fact that β is a subtype of α is expressed by placing β below α in the diagram and drawing a line between them. A large circle represents a collection of types rather than a single type.

Every value has a homomorph in the type GENERAL. This homomorph represents the same computational object as the original value. If a value of any type whatsoever is converted to GENERAL and back again, the resulting value is always identical to the original one. There exist class predicates for INTEGER, BITS, REAL, BOOLEAN, and FUNCTIONAL that are true for values in these respective types, and also for their homomorphs in GENERAL; they are false for all other values. In other words, these class predicates do not distinguish between values and their homomorphs in GENERAL.

There is an obvious correspondence between the different syntactic classes of data and the set of possible types. It should be noted that NODE is a particular n-tuple type. The values corresponding to strings are in type GENERAL. Reading a datum d in a syntactic class α results in a value v which is in type GENERAL, but is a homomorph of a value in the type α' that corresponds to α.

Table 2. Conversion of Type α to Type β

α \ β	B	I	B	R	G	F	A
BOOLEAN	U	X	X	X	H	X	X
INTEGER	T	U	IB	F	H	X	X
BITS	T	BI	U	BR	H	X	X
REAL	T	E	RB	U	H	X	X
GENERAL	GP	GI	GB	GR	U	RH	RH
FUNCTIONAL	T	X	X	X	H	A	X
ARRAY	T	X	X	X	U	X	A

U = setting is unchanged

X = not permitted

T = constant <u>TRUE</u>

H = homomorph requiring change of setting

GI = if homomorph of a number, convert to that number and then to INTEGER; if NIL, then return 0; otherwise illegal

GP = if <u>FALSE</u> then return <u>FALSE</u>, otherwise return <u>TRUE</u>

GB = if homomorph of a number then convert to that number and then to BITS; if NIL, then return 0Q; otherwise illegal

GR = if homomorph of a number, then convert to that number and then to REAL; if NIL, then return 0.0; otherwise illegal

E = apply ENTIER function

F = apply FLOAT function

RH = restricted homomorph; if given value is a homomorph of a value of the desired type, take that value; otherwise illegal

BI = convert to positive integer; illegal if sign bit is negative

IB = convert to bit sequence; -0 treated like +0; other negative numbers illegal

RB = REAL to INTEGER, then INTEGER to BITS

GR = BITS to INTEGER, then INTEGER to REAL

A = types must agree exactly, else illegal

The simple-type corresponding to a given type is the highest node in the lattice of types (as given in Figure 1) that is connected to the given type. In specifying the type of an array, the simple-type that describes its elements is always used. In specifying the type of a function, simple-types are ordinarily used in the valuation-descriptor and in the parameter-descriptors. If a type that is not a simple-type is used in this context, it is semantically equivalent to the corresponding simple-type. If the reference mode is omitted in a parameter-descriptor, DIRECT is assumed; if a reference mode is omitted in a result-descriptor, UNFIELDED is assumed. If a coordinate is omitted with a reference mode of DIRECT or INDIRECT, a standard coordinate, which may depend on the type, is assumed. In the case of the 360, the standard coordinate is (0 . 32), independent of type.

A value is <u>unique</u> if there are no other values with the same type that represent the same computational object. If the values in a type α are unique, then the values in all subtypes of α are unique. A value is <u>invariant</u> if the computational object that it represents cannot be changed by program operations. A value is <u>alterable</u> if it can be made to represent a different computational object through program operations. The data structure associated with an alterable value is also said to be alterable. An alterable data structure consists of a skeleton and a substrate. The <u>skeleton</u> consists of a collection of fields, and possibly settings also, that cannot be changed by program operations. The <u>substrate</u> consists of a collection of settings that can be changed by program operations; these settings are the contents of certain fields of the skeleton.

Values within the types INTEGER, BITS, REAL, BOOLEAN, and FUNCTIONAL are unique. The homomorphs in GENERAL of values within these types are invariant but not unique. The values in SYMBOL are unique, and the homomorphs of these values in GENERAL are also unique. Arrays and n-tuples are alterable, though special cases of them may be invariant. N-tuples and arrays may be constructed either uniquely or non-uniquely. These terms will be explained for the special case of nodes, but the explanation applies equally for any other n-tuple or array type. The procedure for constructing a node has as input a set of values of known type, and as output a single value that represents the desired node. When a node is constructed uniquely, the node construction procedure will always give the same value as output each time it is called with the same set of inputs. When a node is constructed non-uniquely, calling the node construction procedure twice with the same set of values as input will, in general, yield two different values as output.

The substrate of a uniquely constructed n-tuple or array consists of the collected substrates of the values from which it was constructed. The substrate of a non-uniquely constructed n-tuple or array consists of the contents of the fields of that n-tuple or array. Thus an n-tuple or array whose component parts are invariant will itself be invariant. A constant is--by definition--invariant, and possibly unique. It is an error to attempt to modify a constant, though this error will not always be detected.

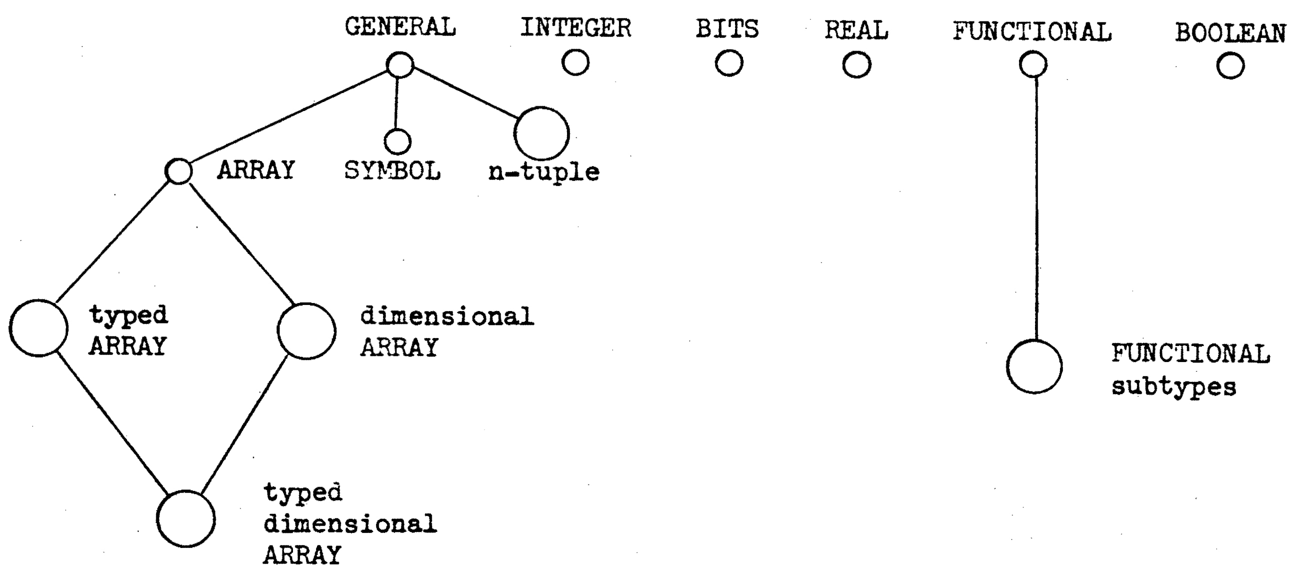


Figure 1. Lattice of Types

2.5 TOKENS AND THEIR SYNTAX

Some of the characters in the kernel language, particularly mark-operators, may be replaced in specific implementations by multiple characters, dot-operators, or reserved words. The special characters of LISP 2 and their meaning are given in Table 3. In token syntax, the term "character" will be defined for specific implementation and the terms "letter", "digit", and "octal-digit" refer to the conventional sub-classes of characters.

T

identifier ≡ literal|genid

literal ≡ letter {letter|digit|.}$_{*}$

character-datum ≡ ¢ character

string ≡ # {non-string-delimiter|'character}$_{*+1}$ #

non-string-delimiter ≡ {any character other than "#", "'", or "R"}

special-spelling ≡ % string

genid ≡ %G string

mark-operator ≡ +|-|*|/|÷|\|↑|→|←|o|∧|∨|∼|=|≠|<|≤|>|≥

boolean ≡ TRUE|FALSE

compound-var-name ≡ identifier$identifier

array-typer ≡ ARRAY$simple-type

punctuator ≡ '|:|;|,|(|)|[|]

sign ≡ +|-

dot-operator ≡ {.}$_{*+1}$ [letter] {letter|digit|.}$_{*}$

decimal ≡ digit$_{*+1}$

unsigned-integer ≡ decimal E decimal|decimal

unsigned-octal ≡ octal-digit$_{*+1}$ Q decimal|octal-digit$_{*+1}$ Q

exponent ≡ E {sign decimal|decimal}

fraction ≡ decimal . decimal|decimal .| . decimal unsigned-real ≡ fraction exponent|fraction

integer ≡ sign unsigned-integer

octal ≡ sign unsigned-octal

real ≡ sign unsigned-real

number ≡ integer|octal|real
|unsigned-integer|unsigned-octal|unsigned-real

token ≡ literal|character-datum|special-spelling
|mark-operator|string|genid|boolean
|array-typer
|compound-var-name|punctuator|sign
|dot-operator|octal|integer|real
|unsigned-octal|unsigned-integer|unsigned-real

In the kernel language, all tokens are required to be separated by at least one space. Spaces may be eliminated between tokens according to the optional transformation summarized in Table 4. In the table, an "X" indicates that the space between α and β is required; no "X" means the space may be eliminated.

3. EXPRESSIONS

An <u>expression</u> designates a computational procedure. The result obtained from carrying out the procedure is a <u>valuation</u>, and the process of carrying out the procedure is called <u>evaluation</u> of the expression. A valuation has one of four <u>reference modes</u>: NOVALUE, UNFIELDED, DIRECT, or INDIRECT. When the valuation has reference mode NOVALUE, the expression is evaluated for its side effects alone and does not produce a value. Valuations in the other three reference modes consist of settings as follows:

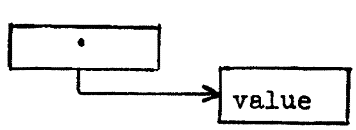
Mode	Setting
UNFIELDED	value
DIRECT	$\boxed{\text{value}}$
INDIRECT	

Table 3. Table of Operators

Kernel language characters and their meaning as SL operators:

+	addition	∧	and
-	subtraction	∨	or
*	multiplication	~	not
/	real division	=	equal
÷	integer division	≠	unequal
\	integer remainder	<	less than
↑	exponentiation	≤	less than or equal
←	assignment	>	greater than
→	loc-assignment	≥	greater than or equal
o	cons	↔	synonym definition
		¢	character-datum

Table 4. Optional Transformations for Elimination of Spaces Between α and β

α \ β		1	2	3	4	5	6	7	8
1	string genid special-spelling punctuator								
2	character-datum								
3	sign					X			X
4	mark-operator				X				
5	dot-operator					X	X	X	X
6	integer octal real					X	X	X	X
7	literal compound-var-name boolean		X			X	X	X	X
8	unsigned-integer unsigned-octal unsigned-real					X		X	X

An "X" indicates that the space between α and β is required; no "X" means the space may be eliminated.

Here a box indicates a field and a pointer to a box indicates a locator of that field. An UNFIELDED valuation consists of a value only. A DIRECT valuation consists of a field containing a value. An INDIRECT valuation consists of a value in a field, a locator of that field, and a second field that contains the locator.

Of the reference modes, NOVALUE is the least restrictive and INDIRECT is the most restrictive. When a valuation in one reference mode appears in a context where a valuation in a different reference mode is required, then reference mode conversion is possible if the desired reference mode is less restrictive than the given one. In this case, the conversion is accomplished simply by discarding the irrelevant parts of the valuation. Thus, in converting from DIRECT to UNFIELDED, the field containing the value is disregarded, and only the setting of the field is considered.

Every field has associated with it a coordinate. The coordinate is (in general) machine-dependent, and consists of two integers: the length of the field in bits, and the position within a computer word of the initial bit. Both DIRECT and INDIRECT valuations have coordinates associated with them, at least implicitly; the coordinate of a DIRECT or INDIRECT valuation describes the field that contains the value. The coordinate of a field containing a locator is standard, and therefore need not be specified.

3.1 CONSTANTS

3.1.1 Syntax of Constants

SL	constant ≡ autonym | 'datum
IL	constant ≡ autonym | (QUOTE datum)
SL IL	autonym ≡ nonsymbol-element | character-datum | special-spelling

3.1.2 Semantics of Constants

A constant has the reference mode UNFIELDED and a value corresponding to the datum that is part of the constant. "QUOTE" or "'" are used in order to distinguish between data and programs. Autonyms are data which, because of their syntax, can never be confused with parts of a program.

3.2 VARIABLES

3.2.1 Syntax of Variables

SL

var-name ≡ untailed-var-name|tailed-var-name
tailed-var-name ≡ compound-var-name
untailed-var-name ≡ first-name

T

compound-var-name ≡ first-name$section-name

IL

var-name ≡ untailed-var-name|tailed-var-name
tailed-var-name ≡ (first-name . section-name)
untailed-var-name ≡ first-name

SL
IL

first-name ≡ identifier
section-name ≡ identifier

3.2.2 Semantics of Variables

A variable is a collection of valuations. These valuations are known as the bindings of the variable. At any time, at most one binding of a variable is said to be active, and this valuation is said to be the <u>active binding</u> of the variable. When a variable is evaluated, its valuation is given by the currently active binding. The contents of the value field of a DIRECT binding or the contents of the locator field of an INDIRECT binding are known as the <u>active assignment</u> of the variable. During the time that a binding is active, the assignment of its variable may change, but the field in which the assignment is to be found will not change.

There are two kinds of variables: lexical variables and section variables. A lexical variable has a first-name associated with it; a section variable has a first-name and a section-name associated with it. A lexical variable is always designated by an untailed-var-name; a section variable may be designed either by a tailed-var-name or an untailed-var-name. At any time, there exists a section-list, which is a list of section-names. The first section-name on the section-list is the current section. Section variables are in one of three categories: PUBLIC, STATIC, or OPTIONAL.

Variables are described by variable declarations, which may appear either in DECLARE statements made on the supervisor level or in block headings and function definitions. Associated with every variable declaration in a block heading or function definition is a lexical scope. The lexical scope of a variable declaration in a block heading consists of the statements within the block; the lexical scope of a variable declaration in a function definition consists of the expression that defines the function. However, if the lexical scope includes a block or function that also declares a variable using the same var-name as the outer declaration, then the lexical scope of the outer declaration excludes the lexical scope of the inner one.

When an expression is in the form of a var-name, then the following rules, applied sequentially at compile time, determine the variable to which it refers:

1. If the var-name lies within the lexical scope of a variable declaration using that var-name, then the var-name refers to that variable.

2. If the var-name is tailed, then it refers to a section variable in the section named by the section-name.

3. If a section variable whose first-name is the same as the var-name exists in any section on the section list, then the var-name refers to the one of these whose section appears first on the section list.

If none of these rules apply, then an error condition exists.

The active binding of a variable is determined at run time. A DECLARE statement establishes an active binding for each variable that it declares, and this binding never disappears thereafter, though it may be superseded temporarily. Upon entering a block or function, active bindings are established for each variable declared by the block or function, and the previous active bindings of these variables become inactive. The nature of each of these new bindings is determined by its variable declaration. Upon exit from a block or function, the bindings created by it disappear.

A variable may have any of the four reference modes. However, the reference mode UNFIELDED is always used for variables whose values are fixed function definitions, and the reference mode NOVALUE is always used for macros. From the restrictions on assignments, it then follows that assignments cannot be made to variables denoting either fixed functions or macros. Furthermore, variables denoting macros cannot be evaluated.

3.3 SIMPLE EXPRESSIONS

3.3.1 Syntax of Simple Expressions

SL

```
expression ≡ simple-expression|assignment_expression
              |funarg|conditional-expression

assignment-expression ≡ {var-name|form} {←|→} expression

simple-expression ≡ disjunction

disjunction ≡ conjunction {∨ conjunction}*

conjunction ≡ negation {∧ negation}*

negation ≡ relation|∿ negation

relation ≡ construct {relator construct}*

relator ≡ =|≠|<|≤|>|≥

construct ≡ sum {o sum}*

sum ≡ [+|-] term {{+|-} term}*

term ≡ factor {{*|/} factor}*

factor ≡ power|factor {\|÷} power

power ≡ primary [↑ power]

primary ≡ basic-expression|(expression)|block-expression

form ≡ operator ({operand}*, )

operator ≡ var-name
```

IL

```
expression ≡ basic-expression|conditional-expression|block-expression
              |funarg

form ≡ (operator operand*)

operator ≡ mark-operator|var-name
```

SL
IL

```
operand ≡ expression

funarg ≡ function-definition

basic-expression ≡ constant|var-name|form
```

3.3.2 Semantics of Simple Expressions

The syntactic hierarchy for nonconditional-expressions in SL specifies the hierarchy of the various operators, and thus the implicit parenthesization. The parenthesization is explicit in IL. The semantics of SL expressions employing operators are specified by giving the translation of these expressions into IL and then giving the semantics of the corresponding forms in IL.

A disjunction

$$c_1 \vee c_2 \vee \dots \vee c_n$$

in SL translates into

$$(\vee\ c_1'\ c_2'\ \dots\ c_n')$$

in IL where c_i' is the IL translation of c_i. The translations of conjunctions and constructs are analogous. A negation $\sim n$ in SL translates into $(\sim n')$ in IL. A relation

$$c_1\ r_1\ c_2\ r_2\ \dots\ r_n\ c_{n+1}$$

in SL translates into

$$(\text{RELATION}\ c_1'\ r_1\ c_2'\ r_2\ \dots\ r_n\ c_{n+1}')$$

in IL. A sum in SL translates into a form whose operator is "+". A term t preceded by "+" translates into t'; a term t preceded by "-" translates into (- t'). The translation of a term is analogous, with "*" behaving like "+" and "/" behaving like "-". The operators "→", "←", "÷", "\", and "↑" are all binary, and the parse in SL determines their operands. The SL expression a x b, where a and b are operands and x is a binary operator, translates into (x a' b') in IL.

The operator of a form must designate either a function, an array, or a macro. If the operator designates a function, then the form is evaluated as follows:

1. The operands of the form are evaluated. The order of evaluation is not guaranteed.

2. A binding is created for each variable that is a parameter of the function. The valuations obtained in Step 1 are then assigned to the corresponding variables. None of these assignments are performed until all of the arguments are evaluated. If a parameter is DIRECT, the rules for ordinary assignment apply; if the parameter is INDIRECT the rules for loc-assignment apply.

3. The bindings created in Step 2 are activated and the expression defining the function is evaluated. A valuation is obtained unless the result-declaration of the function is NOVALUE. After the evaluation is completed, the bindings created in Step 2 disappear.

4. If the valuation obtained in Step 3 is of the same type, reference mode, and coordinate as was declared for the function, then this valuation is the valuation of the entire form. If the reference mode declared for the function is NOVALUE, then no valuation is obtained for the form. If the reference mode is UNFIELDED, then type conversion will be performed if the type of the valuation obtained from the expression does not agree with the type declared for the function. If the reference mode is DIRECT or INDIRECT, then the type and coordinate of the valuation obtained from the expression must agree with the type and coordinate declared for the function or an error condition exists.

If the operator designates an array, then the values of the successive operands are its subscripts. The valuation of the form has reference mode DIRECT and the type subspecified for the array. The valuation consists of the field containing the designated element within the array. The subscripts are evaluated in an unspecified order and converted to type INTEGER in order to find the desired array element.

If the operator designates a macro, then the macro specifies a transformation to be applied to the form at compile time. The transformed form replaces the original one. Macros always operate on the IL version of a form. In order to determine the valuation of a form whose operator is a macro, the explanation of that specific macro must be consulted. Although macros are variables, their reference mode is NOVALUE and therefore they can only be used in the context of form operators. Assignments cannot be made to them, nor can they be evaluated by themselves or passed as functional arguments.

Assignment expressions are used in order to transmit a valuation from one field to another. They are also used implicitly in the transmission of arguments to functions. The transmission occurs as a side effect of the evaluation of the assignment expression; the valuation of either kind of assignment expression is always the valuation of its second operand. There are two kinds of assignment expressions: ordinary assignment expressions and loc-assignment expressions. The operator "←" designates ordinary assignment and the operator "→" designates

Table 5. Semantics of Assignment Expressions

context / reference mode	$\alpha \leftarrow \underline{\beta}$	$\underline{\alpha} \leftarrow \beta$	$\alpha \rightarrow \underline{\beta}$	$\underline{\alpha} \rightarrow \beta$
NOVALUE	X	X	X	X
UNFIELDED	1	X	3	X
DIRECT	1	2	4	X
INDIRECT	1	2	5	6

X = not permitted

1. Obtain the value of β. Convert it to type α if such conversion is permissible; illegal otherwise. Valuation is UNFIELDED.

2. Place value obtained from β in the value field of α.

3. Valuation is a locator of a newly obtained field with standard coordinate whose setting is the value of β.

4. α and β must have the same type and the same coordinate. The valuation β is a locator of the field containing the value of β.

5. α and β must have the same type and the same coordinate. The valuation is the locator that locates the field containing the value of β.

6. Replace the locator part of α by the valuation of β, which is also a locator, i.e., put the contents of the locator field of β into the locator field of α.

loc-assignment. An ordinary assignment expression transmits a value only; a loc-assignment expression transmits a locator of a field containing a value. The most useful and common case of loc-assignment is the transmission of a binding of a variable. If A and B are variables, evaluation of the loc-assignment expression A → B causes the active binding of B to be transmitted to A (which must have reference mode INDIRECT). Thereafter, any change to the value of B will cause the same change to be made to the value of A, and conversely.

The semantics of assignment expressions are given in greater detail in Table 5. In order to determine the effect of an assignment expression, the reference modes of its operands must be known. First, the table is used to determine the valuation obtained from the second operand. This is done by looking in the column corresponding to the second operand of the appropriate kind of assignment expression and the row corresponding to the reference mode of that operand. Then the column corresponding to the first operand of the assignment expression and the row corresponding to the reference mode of this operand is used to determine what is done with the valuation obtained from the second operand.

When a value is copied from one field to another, the new value is identical to the old one. Consequently, both values have the same substrate. A change to the substrate of a value changes the substrate of all identical values in the same way. Thus, for instance, if a nonuniquely constructed array is transmitted through assignment from one variable to another, changes to the elements of the array via one of these variables will change the datum represented by the value of the other variable. Furthermore, if a constant is assigned to a variable, then the value of that variable is invariant and any attempt to modify its associated data structure is an error.

The form ($\wedge$ p_1 p_2 ... p_n) has value FALSE if any p_i has value FALSE, and TRUE otherwise. The expression is evaluated from left to right only far enough to determine its value, i.e., if any p_i is false, the remaining p_j for $j > i$ are not evaluated. ($\wedge$) has value TRUE.

The form ($\vee$ p_1 p_2 ... p_n) has value TRUE if any p_i has value TRUE, and FALSE otherwise. The expression is evaluated from left to right only far enough to determine its value, i.e., if any p_i is TRUE, then the remaining p_j for $j > i$ are not evaluated. ($\vee$) has value FALSE.

A form (CASE s e_1 e_2 ... e_n) is evaluated by evaluating s to obtain an integer x. If this integer is in the range $1 \le x \le n$, then the value of the entire CASE expression is e_x; otherwise the value is e_n.

A form whose operator is RELATION causes a sequence of tests to be performed to yield a BOOLEAN value. The tests are performed from left to right as specified by the relations; if all the tests are satisfied the result is TRUE, and otherwise the result is FALSE. The evaluation proceeds only far enough to determine the value of the RELATION expression. It is guaranteed that none of the quantities in the relation are evaluated more than once.

The arithmetic operations "+" and "*" are carried out by means of macros. The type of the value obtained from the operation is determined by the following rules, applied sequentially:

1. If all of the operands are of type INTEGER or BITS, then the result is of type INTEGER.

2. If none of the operands are of type GENERAL, then the result is of type REAL.

3. If none of the operands are of type REAL or homomorphs in GENERAL of type REAL, the result is of type GENERAL and is a homomorph of a value of type INTEGER.

4. Otherwise, the result is of type GENERAL and is a homomorph of a value of type REAL.

The operator "-" expects a single argument. If the argument is BITS or INTEGER the result is INTEGER, otherwise REAL. The operator "/" expects a single argument of type REAL, and forms its reciprocal, also of type REAL. The operators "÷" and "\" have arguments and value of type INTEGER.

3.4 CONDITIONAL EXPRESSIONS

3.4.1 Syntax of Conditional Expressions

SL conditional-expression ≡ {IF predicate THEN consequent ELSE}$_{*+1}$ {terminal-expression|Λ}

RT ELSEΛ{$\overline{\text{ELSE}}$} → empty {$\overline{\text{ELSE}}$}

RT ELSE Λ ELSE → ELSE ELSE

OT ELSE IF ⊂ conditional-expression → IF

IL	conditional-expression ≡ (IF {predicate consequent}$_*$ [terminal-expression])

SL	predicate ≡ expression
	consequent ≡ expression
	terminal-expression ≡ expression

3.4.2 Semantics of Conditional Expressions

The symbol "$\wedge$" is used in the SL kernel language for conditional-expressions in order to force explicit indication of the grouping of the parts of nested conditional-expressions. In translating to IL, a "$\wedge$" at the end of a conditional-expression indicates that the terminal-expression is omitted.

In evaluating a conditional-expression, the predicates are evaluated in turn and the resulting values are converted to BOOLEAN. As soon as one of the evaluations results in TRUE, the corresponding consequent is evaluated, and its valuation becomes the valuation of the entire conditional-expression. Neither the predicates following the first true predicate nor the consequents other than the one corresponding to the first true predicate are evaluated. If none of the predicates are true, then the valuation of the conditional-expression is the valuation of its terminal-expression, if there is one. If none of the predicates are true and there is no terminal-expression, then an error condition exists.

If all of the consequents and the terminal-expression of a conditional-expression have the same type, then the type of the valuation of the conditional-expression is that type. Otherwise the type of the valuation of the conditional-expression is GENERAL, and type conversion is performed when the conditional-expression is evaluated. The reference mode is the most restrictive of the reference modes of the various consequents and the terminal-expression.

3.5 FUNCTION DEFINITIONS

3.5.1 Syntax of Function Definitions

SL

```
function-definition ≡ FUNCTION parameter-list [result-declaration]
                      {; parameter-declaration}*: body

parameter-list ≡ ({parameter}*, [, indef-parameter])
        RT  (, indef-parameter) ⇒ (indef-parameter)

parameter-declaration ≡ {parameter}*+1, {[variable-reference-mode]||
                         [parameter-storage-mode]||
                         [type-declaration]||[coordinate]}

indef-parameter ≡ var-name (indef-name)

type-declaration ≡ type|LIKE var-name

result-declaration ≡ {[variable-reference-mode]||
                      [type-declaration]||[coordinate]}
                      |NOVALUE|{UNFIELDED||[type-declaration]}
```

IL

```
function-definition ≡ (FUNCTION[result-declaration]
                      parameter-declaration-list body)

result-declaration ≡ ([reference-mode] [type-declaration]
                      [coordinate])|NOVALUE|(UNFIELDED
                      [type-declaration])

type-declaration ≡ type|(LIKE var-name)
        OT  (type) ⊂ result-declaration ⇒ type

parameter-declaration-list ≡ (single-parameter-declaration*
                              [indef-parameter-declaration])

single-parameter-declaration ≡ parameter|(parameter
                  [type-declaration] [parameter-storage-mode]
                  [variable-reference-mode] [coordinate])

indef-parameter-declaration ≡ (indef-name [type-declaration]
                  [variable-reference-mode] INDEF parameter)
```

SL IL

parameter ≡ var-name

indef-name ≡ untailed-var-name

parameter-storage-mode ≡ LEXICAL|PUBLIC

body ≡ expression

3.5.2 Semantics of Function Definitions

A function-definition is an expression whose value is a function. The function does not have a name associated with it unless the function-definition appears as part of a named-function-definition in a DECLARE statement on the top level. Because of the behavior of free variables in a function-definition, evaluation of a function-definition at different times may yield effectively different functions.

At run time, a function-definition is subject to two processes: evaluation and application. These two processes occur at different times and in different contexts. A function-definition may be evaluated once and applied many times; application occurs whenever a FUNCTIONAL variable whose value is the function appears as the operator of a form which is being evaluated.

The variables named within a function-definition can be divided into three groups:

1. Variables that are used as parameters of the function or are declared in a block within the function-definition (bound variables).

2. Variables that are not parameters of the function but are lexically bound from outside the function-definition (funarg-variables). A variable in this group is declared in a block or function-definition that contains the function-definition in question.

3. Variables that do not lie within the lexical scope of any declaration (free variables).

A function-definition on the top level does not have any outer lexical context and hence can only have bound variables and free variables.

At the time of evaluation of a function-definition, the valuation of each of its funarg-variables is obtained and saved. During the application of the function, new bindings are created for the funarg-variables, and the preserved valuations are assigned to these bindings. These bindings are created after the arguments of the function are evaluated, but before the body of the function is evaluated; they disappear after the evaluation of the body of the function. The bound variables of the function receive the valuations obtained from the arguments, and the free variables of the function obtain the valuations given by their currently active bindings.

An unnamed function prints out as "[FUNCTION]"; this form does not correspond to any datum, and hence cannot be read back in.

The parameter-declarations of a function-definition are treated like the block-variable-declarations for a block, except that the parameters of a function do not have presets. Both the rules for defaulting of attributes and the rules for determining the variable designated by a given var-name are the same. The result-declaration determines the type, reference mode, and coordinate of the valuation returned by the function; since these attributes describe a valuation rather than a variable, no storage mode is required.

If the parameter-list for a function-definition contains an indef-parameter, then the function expects an indefinite number of arguments. In matching arguments against parameters when the function is applied, the ordinary parameters, if any, are matched first. The remaining parameters constitute the indef-argument, which is a list of valuations. The length of this list is assigned to the indef-name upon entrance to the function; the indef-name is implicitly assumed to be a LEXICAL INTEGER DIRECT variable with standard coordinate. The list of valuations is formed into a one-dimensional array, and this array is then assigned to the var-name of the indef-parameter. Within the body of the function-definition, then, the indef-arguments of the function are treated as elements of this array, and the indef-name gives the length of this array.

3.6 BLOCK EXPRESSIONS

3.6.1 Syntax of Block Expressions

SL IL	block-expression ≡ begin-block|do-block

3.6.2 Semantics of Block Expressions

The valuation of a block-expression is obtained by executing the block-expression according to the rules for block execution until an implicit or explicit return-statement belonging to the block-expression is encountered. The valuation associated with this return-statement then determines the valuation of the entire block-expression. If there is more than one return-statement belonging to a given block-expression, then the type of the valuation of the block-expression is the same as the type of the valuations of the return-statements if they are all the same, and GENERAL otherwise. The reference mode of the valuation is the most restrictive of the reference modes of the valuations of the return-statements. If the valuations of the return-statements differ in coordinate, however, the valuation of the block-expression will have reference mode UNFIELDED.

The labels belonging to a block-expression are not visible outside the block-expression.

4. BLOCKS

4.1 SYNTAX OF BLOCKS

SL

block ≡ begin-block|do-block

do-block ≡ DO {statement ;}$_*$ END

begin-block ≡ BEGIN {block-variable-declaration}$^{;}_{*}$:
{statement ;}$_*$ END

block-variable-declaration ≡ block-variable-preset|
block-variable-attribution

block-variable-preset ≡ {block-variable [{←|→} preset]}$^{,}_{*+1}$

block-variable-attribution ≡ {block-variable}$^{,}_{*+1}$ attribute$_{*+1}$

attribute ≡ variable-reference-mode|parameter-storage-mode|
type-declaration|coordinate

statement ≡ label$_*$ [unlabeled-statement]

OT ; END ⊂ block ⟹ END

label ≡ identifier :

IL

block ≡ begin-block|do-block

begin-block ≡ (BEGIN (block-variable-list) statement$_*$)

do-block ≡ (DO statement$_*$)|(BEGIN nil statement$_*$)

block-variable-list ≡ {(block-variable [type-declaration]
[parameter-storage-mode] [variable-reference-mode]
[coordinate] [preset])}$_{*+1}$

OT (block-variable) ⊂ block-variable-list ⟹ block-variable

statement ≡ label$_*$ statement-unit|label

statement-unit ≡ (LABEL label statement-unit)|unlabeled-statement

label ≡ identifier

SL
IL

block-variable ≡ var-name

preset ≡ expression

$$\text{unlabeled-statement} \equiv \left\{\begin{array}{l} \text{compound-statement} \\ \text{block-statement} \\ \text{go-statement} \\ \text{conditional-statement} \\ \text{case-statement} \\ \text{casego-statement} \\ \text{return-statement} \\ \text{for-statement} \\ \text{code-statement} \\ \text{try-statement} \\ \text{expression} \end{array}\right\}$$

compound-statement ≡ do-block|(compound-statement)

block-statement ≡ begin-block|(block-statement)

4.2 SEMANTICS OF BLOCKS

A block may appear in either a statement context or an expression context. A block is evaluated in the same way, independent of the context in which it appears; however, a block appearing in expression context produces a valuation, while a block appearing in statement context does not.

The evaluation of a block proceeds in the following manner:

1. A binding is created for each block-variable, and an initial valuation is assigned to this binding. The initial valuation is determined by evaluating the preset, if it is given; otherwise it is determined by default. None of these assignments are performed until all of the presets have been evaluated; if any of the block-variables appear as part of one of the presets, the previously active binding of such a block-variable is used. If a block-variable is DIRECT, then the rules for ordinary assignment are used. (In the context of a block-variable-preset, no distinction is made between "→" and "←"; the method of assignment is determined strictly from the reference-mode of the block-variable.)

2. The statements in the block are executed until a termination occurs. Termination will result from execution of a GO statement whose label lies outside the block; from evaluation of an EXIT expression; from execution of a RETURN statement; or from execution of the last statement of the block without a transfer of control. (There is always an implicit "RETURN NIL" statement at the end of a block.) The statements of the block are executed sequentially unless a transfer of control or a block termination occurs. When control is transferred to a label, the statement execution sequence continues with the statement following that label.

3. After termination, the bindings created upon block entrance disappear.

The declarations for a block-variable specify, either implicitly or explicitly, four attributes and a preset. The four attributes are: reference mode, storage mode, type, and coordinate. For a block specified in SL, the block-variables are determined by collecting them from the block-variable-declarations. A block-variable may appear in any number of block-variable-declarations, and the attributes of a block-variable are collected from all declarations in which it appears. If any of the attributes are contradictory, an error condition exists. For a block specified in IL, each block-variable is given just once in the block-variable-list, and its attributes and initial valuation appear along with it. In either case, omitted attributes and presets are determined by default.

If the reference mode of a block-variable is not given, it is defaulted to DIRECT. If the storage mode is not given, it defaults to PUBLIC if the block-variable is represented either by a tailed-var-name or by an untailed-var-name whose first name is the same as that of a section-variable of the current section. Otherwise the storage-mode defaults to LEXICAL. If preset is given explicitly, then the default type is the type of the preset; otherwise, the default type of the current section is used. The default coordinate is the standard one for the type of the block-variable.

If a block-variable is tailed, then it designates a section variable of the section named by the section-name; otherwise, the variable that it designates depends upon the storage-mode of the block-variable. If the storage-mode is PUBLIC (either explicitly or by default), then the block-variable designates a section variable of the current section whose first-name is the same as the var-name; otherwise, the block-variable designates a lexical variable.

If the type of a block-variable is determined by LIKE, then the associated var-name must either be a tailed-var-name or must agree with a PUBLIC or STATIC section-variable in the current section. In this case, the type of the block-variable is taken to be the same as that of the section variable named by the var-name.

The type, reference mode, and coordinate of a block-variable which is a section variable must be the same as those specified in the top-level declaration of the section variable.

If a preset is not explicitly given for a block-variable, then it is determined by default from the type of the block-variable. The standard default presets are as follows:

Type	Preset
GENERAL	nil
INTEGER	0
BITS	0Q
REAL	0.0
BOOLEAN	FALSE
Any functional-type	error trap
Any n-tuple type	n-tuple with default component values
Dimensional, typed array	array as specified, with elements defaulted according to type of array
Dimensional array	GENERAL ARRAY with nil elements
Other	nil

Every block has a set of labels (possibly empty) that belong to that block. The labels that belong to the block consist of those labels that are directly attached to the statements of the block, plus those labels that belong to any conditional-statement or compound-statement of the block. There must be no duplications among the labels that belong to a block. The labels that belong to a block-expression or a block-statement within an outer block do not belong to the outer block.

In parsing the statements of a block, the possible parses are considered in the order in which they appear in the syntax equation for unlabeled-statement. It therefore follows that a statement will be considered as an expression if and only if it cannot be considered as any other kind of statement. When an expression is encountered in a context where a statement is expected, the expression is evaluated and the valuation is discarded. An expression appearing in statement context may have the reference mode NOVALUE.

4.3 COMPOUND-STATEMENTS AND BLOCK-STATEMENTS

When a block appears in a statement context, it is either a compound-statement or a block-statement, and never a block-expression. A block enclosed in parentheses is treated exactly as though the parentheses were removed.

A block-statement declares at least one block-variable, while a compound-statement does not declare any block-variables. The labels within a compound-statement are visible from outside the compound-statement, while the labels within a block-statement are not visible from outside the block-statement. Hence the labels that belong to a compound-statement are not allowed to duplicate the labels that belong to the block containing the compound-statement, but the labels that belong to a block-statement are allowed to duplicate the labels that belong to the block containing it.

4.4 GO STATEMENTS

4.4.1 <u>Syntax of GO Statements</u>

SL	go-statement ≡ GO label
IL	go-statement ≡ (GO label)

4.4.2 Semantics of GO Statements

A go-statement causes a transfer of control to its label. The label is found by the following algorithm:

1. Let the scope of the go-statement be the innermost block-statement or block-expression that contains the go-statement.

2. If the label belongs to the scope, then the go-statement transfers control to that label.

3. If the scope is a block-expression or there is no block surrounding the scope, then the label is undefined and an error condition exists.

4. Let the new scope be the innermost block-statement or block-expression that surrounds the old scope, and return to Step 2.

4.5 RETURN STATEMENTS

4.5.1 Syntax of Return Statements

SL	return-statement ≡ RETURN expression
IL	return-statement ≡ (RETURN expression)

4.5.2 Semantics of Return Statements

A return-statement causes termination of the innermost block-expression containing it, and thus also causes termination of all block-statements and compound-statements that contain the return-statement and are contained within this block-expression. As each of these blocks is terminated, the bindings of its block-variables disappear. A return-statement produces a valuation obtained by evaluating the expression that is part of the return-statement. If it is desired to terminate a block-statement or compound-statement without terminating the block or blocks that surround it, then an explicit transfer of control to the end of the block-statement or compound-statement should be used.

4.6 CONDITIONAL STATEMENTS

4.6.1 Syntax of Conditional Statements

SL

conditional-statement ≡ {IF predicate THEN statement-consequent ELSE}$_{*+1}$ [terminal-statement|Λ]

RT ELSE Λ {$\overline{\text{ELSE}}$} $\Longrightarrow$ {$\overline{\text{ELSE}}$}

OT ELSE IF $\subset$ conditional-statement $\Longrightarrow$ IF

statement-consequent ≡ statement

terminal-statement ≡ statement

IL

conditional-statement ≡ (IF {predicate statement-consequent}$_*$ [terminal-statement])

statement-consequent ≡ statement-unit

terminal-statement ≡ statement-unit

4.6.2 Semantics of Conditional Statements

In translating an SL conditional-statement to IL, a "Λ" at the end of the conditional-statement indicates that the terminal-statement is to be omitted. In order to evaluate a conditional-statement, the successive predicates are evaluated in turn, and the resulting values are converted to BOOLEAN. As soon as one of the evaluations results in TRUE, the corresponding statement-consequent is executed. Unless this execution causes a transfer of control, control then passes to the next statement in the block. If none of the predicates evaluate to TRUE, then the terminal-statement is executed. If there is no terminal-statement or the terminal-statement is "Λ", then no action is taken and control passes to the next statement in the block.

It is permissible to transfer control to a label within a conditional-statement; the effect is the same as if control had reached the label through normal execution of the conditional-statement. The labels that belong to a conditional-statement are the labels that belong to its statement-consequents and to its terminal-statement. The labels that belong to a conditional-statement also belong to the block that contains the conditional-statement.

4.7 FOR STATEMENTS

4.7.1 Syntax of FOR Statements

SL

$$\textbf{for-statement} \equiv \text{FOR} \left\{ \begin{array}{l} \textbf{reset-clause} \\ \textbf{in-clause} \\ \text{on-clause} \\ \text{step-clause} \end{array} \right\}^{;}_{*} \left\{ \begin{array}{l} \text{unless-clause} \\ \text{while-clause} \end{array} \right\}_{*} : \text{ iterand}$$

unless-clause ≡ UNLESS **expression**

while-clause ≡ WHILE **expression**

reset-clause ≡ **variable** [← **initializer**] RESET **expression**

loop-clause ≡ **variable** LOOP **expression**

in-clause ≡ **variable** IN **expression**

on-clause ≡ **variable** ON **expression**

step-clause ≡ **variable** [← **arithmetic-initializer**] STEP
arithmetic-expression [UNTIL relator
arithmetic-expression]

IL

$$\textbf{for-statement} \equiv \text{(FOR} \; (\left\{ \begin{array}{l} \text{reset-clause} \\ \text{in-clause} \\ \text{on-clause} \\ \text{step-clause} \end{array} \right\}_{*} \left\{ \begin{array}{l} \text{unless-clause} \\ \text{while-clause} \end{array} \right\}_{*}) \text{ iterand)}$$

unless-clause ≡ (UNLESS **expression**)

while-clause ≡ (WHILE **expression**)

reset-clause ≡ (RESET **variable initializer expression**)

in-clause ≡ (IN **variable expression**)

on-clause ≡ (ON **variable expression**)

step-clause ≡ (STEP **variable arithmetic-initializer**
arithmetic-expression [relator arithmetic-expression])

loop-clause ≡ (LOOP variable expression)

SL IL	iterand ≡ unlabeled-statement
	variable ≡ var-name

4.7.2 Semantics of FOR Statements

Each kind of clause has four attributes associated with it: a set of temporary variables, a set of initializations, a test, and a modification. The variables that are initialized may or may not be temporary. Any of the attributes may be null. A for-statement generates the following block:

```
BEGIN initializations
      lab1: tests
            iterand
      lab3: modifications
            GO lab1
      lab2: END
```

A for-statement with no clauses is equivalent to the iterand by itself. The iterand is implicitly surrounded by a begin-block, so that labels within the iterand are not visible outside of the for-statement. The labels lab1 and lab2 are genids.

The attributes of the various for-clauses are as follows:

1. Reset-clause

Temporary variables:	None
Initialization:	Set variables to initializer, if there is one; otherwise none
Test:	None
Modification:	Set variables to expression

2. <u>In-clause</u>

Temporary variables:	List iterator, called gl
Initialization:	Set gl to the expression; set variable to car(gl) if defined, NIL otherwise
Test:	If null(gl) then go to lab2
Modification:	Set gl to cdr(gl); set variable to car(gl) if gl not NIL

3. <u>On-clause</u>

Temporary variables:	None
Initialization:	Set variable to expression
Test:	If null(variable) then go to lab2
Modification:	Set variable to cdr(variable)

4. <u>Step-clause</u>

Temporary variables:	Increment gl and terminator g2
Initialization:	Set gl to first expression; set g2 to second expression if it exists; set variable to initializer if it exists
Test:	If there is a relator part, and if (variable relator g2) is satisfied, then go to lab2
Modification:	Set variable to variable + gl

5. <u>While-clause</u>

Temporary variables:	None
Initialization:	None
Test:	If boolean value of expression is <u>FALSE</u>, then go to lab2
Modification:	None

6. Unless-clause

Temporary variables:	None
Initialization:	None
Test:	If boolean value of expression is TRUE, then go to lab3
Modification:	None

An example of a for-statement and the equivalent block it generates are given below:

```
FOR X IN L1; Y IN L2; I ← 1 STEP 1 >

    LENGTH (X) UNLESS ∿ NUMBERP(X) ∨

    ∿  NUMBERP(Y): SUM ← I*X*Y + SUM

BEGIN G1 ← L1, G2 ← L2, G3 ← LENGTH(X); G1, G2 GENERAL; G3 INTEGER:

    X ← IF NULL (G1) THEN NIL ELSE CAR(G1);

    Y ← IF NULL (G2) THEN NIL ELSE CAR(G2);

    I ← 1;

L1: IF NULL (G1) THEN GO L2;

    IF NULL (G2) THEN GO L2;

    IF I > G3 THEN GO L2;

    IF ∿ NUMBERP(X) ∨ ∿ NUMBERP(Y) THEN GO L3;

    SUM ← I*X*Y + SUM;

L3: G1 ← CDR(G1);

    IF ∿ NULL (G1) THEN X ← CAR(G1);

    G2 ← CDR(G2);

    IF ∿ NULL (G2) THEN Y ← CAR(G2)

    I ← I + 1;

    GO L1;

L2: END
```

4.8 CASE AND CASEGO STATEMENTS

4.8.1 Syntax of CASE and CASEGO Statements

SL

case-statement ≡ CASE (subscript, $\{\textbf{statement}\}^{,}_{*+1}$)

casego-statement ≡ CASEGO (subscript, $\{\text{label}\}^{,}_{*+1}$)

IL

case-statement ≡ (CASE subscript $\textbf{statement}_{*+1}$)

casego-statement ≡ (CASEGO subscript label_{*+1})

SL
IL

subscript ≡ expression

4.8.2 Semantics of CASE and CASEGO Statements

A case-statement is executed by first evaluating the subscript and converting the result to INTEGER to obtain an integer x. If x is in the range $1 \le x \le n$, where n is the number of statements following the subscript, then the xth statement is executed. Otherwise the last statement is executed. Labels within a statement of a case-statement are not accessible from outside the case-statement nor from within the other statements of the case-statement.

A casego-statement is exactly equivalent to the case-statement where each label in the casego-statement is replaced by GO x in SL or (GO x) in IL.

4.9 TRY STATEMENTS

4.9.1 Syntax of TRY Statements

SL

try-statement ≡ TRY var-name; statement; statement

IL

try-statement ≡ (TRY var-name statement statement)

4.9.2 Semantics of TRY Statements

The semantics of a try-statement depends upon the semantics of the operator EXIT. EXIT acts like a function of one argument; its valuation is simply the valuation of its argument converted to GENERAL UNFIELDED.

Execution of a try-statement begins with execution of its first statement. If no EXIT expression is encountered during the execution, control passes to the statement following the try-statement. Otherwise the variable designated by the var-name is set to the valuation of the EXIT expression (using ordinary assignment) and the second statement within the try-statement is executed. Control then passes to the statement following the try-statement. Labels within the try-statement are not accessible outside of the try-statement.

4.10 CODE STATEMENTS

4.10.1 Syntax of CODE Statements

SL	code-statement ≡ CODE ($item_*$)
IL	code-statement ≡ (CODE $item_*$)

4.10.2 Semantics of CODE Statements

The successive items correspond to successive labels and instructions in a LAP program. The syntax of item is defined in the LAP specification document. The effect of a code-statement is to cause execution of the LAP code that it represents.

5.0 DECLARE STATEMENTS ON THE TOP LEVEL

5.1 SYNTAX OF DECLARE STATEMENTS

SL

declare-statement ≡ DECLARE $\{$top-level-declaration;$\}_*$;

top-level-declaration ≡ section-variable-declaration
|named-function-definition
|n-tuple-definition|synonym-definition

section-variable-declaration ≡ section-variable-preset
|section-variable-attribution

section-variable-preset ≡ $\{$section-variable [$\{\leftarrow|\rightarrow\}$ preset]$\}^{,}_{*+1}$

section-variable-attribution ≡ $\{$section-variable$\}^{,}_{*+1}$ $\text{section-attribute}_{*+1}$

section-attribute ≡ variable-reference-mode|section-storage-mode
|type-declaration|coordinate

named-function-definition ≡ section-variable function-definition

synonym-definition ≡ section-variable ↔ expression

n-tuple-definition ≡ n-tuple-type NTUPLE ({coordinate-spec}$^{,}_{*+1}$)

coordinate-spec ≡ type [coordinate]

IL

declare-statement ≡ (DECLARE top-level-declaration$_{*}$)

top-level-declaration ≡ named-function-definition
|section-variable-declaration
|n-tuple-definition
|synonym-definition

section-variable-declaration ≡ section-variable
|(section-variable [type-declaration]
[section-storage-mode]
[variable-reference-mode]
[coordinate] [preset])

OT (section-variable) ⟹ section-variable

named-function-definition ≡ (section-variable FUNDEF
function-definition)

synonym-definition ≡ (section-variable ↔ expression)

n-tuple-definition ≡ (n-tuple-type NTUPLE (coordinate-spec$_{*+1}$))

coordinate-spec ≡ (type [coordinate])

SL
IL

section-variable ≡ var-name

section-storage-mode ≡ STATIC|OPTIONAL|PUBLIC

5.2 SEMANTICS OF DECLARE STATEMENTS

A top-level declaration is used to create a section-variable or modify the attributes of an existing section-variable. If a section-variable is in the form of an untailed-var-name, it creates or modifies a section-variable in the current section; otherwise it creates or modifies a section-variable in the section named by the section-name. Once a section-variable has been declared, only its assignment may be changed, unless there are not yet any references to that variable from assembled code or synonyms.

The top-level-declarations in a single DECLARE statement are assumed to be performed simultaneously, just like the initialization of block-variables. The meaning of section-variable-presets and section-variable-attributions is like that of block-variable-presets and block-variable-attributions, with the following exceptions:

1. There are two additional storage modes: STATIC and OPTIONAL. A STATIC variable cannot be declared PUBLIC in any block or function-definition, but it may be used as a free variable of a block or function-definition.

   When a section-variable is declared OPTIONAL, then a declaration of a block-variable or function-definition with the same first-name and no section-name will refer to the section-variable if it is explicitly declared PUBLIC at the point of declaration, and to a LEXICAL variable otherwise.

2. If a section-variable-preset refers to a section-variable established by a prior DECLARE statement, then the type of the preset does not affect the type of the section-variable, and any necessary type-conversions are performed.

A named-function-definition establishes a section-variable whose reference-mode is UNFIELDED, whose type is the same as the type of the function defined by the function-definition, and whose valuation is the function defined by the function-definition. If this valuation is printed out, the name of the section-variable will appear, and the resulting datum can be read back in. The valuation of such a section-variable can only be modified by a subsequent named-function-definition. A named-function-definition, being on the top level, cannot have any funarg variables.

A synonym-declaration causes the expression on the right to be substituted for the section-variable on the left whenever that section-variable is referred to in a compiled expression or function-definition.

An n-tuple-definition establishes a new kind of n-tuple whose successive fields are described by the coordinate-specs of the n-tuple-definition. For each field, the type and coordinate of the value to be placed there are specified. N-tuples cannot contain locators in their fields, so no reference mode need be given.

Index to Syntax Equations

SL

Index to Syntax Equations (Cont.)

SL
IL

Index to Syntax Equations (Cont.)

IL

Index to Syntax Equations (Cont.)

T