This document was produced by SDC in performance of contract SD-97

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Ave. / Santa Monica, California

TM- 2260/002/01

AUTHOR S. L. Kameny

TECHNICAL

RELEASE C. Baum

C. Baum

for

D. L. Drukey

DATE 9/ 3/65 PAGE 1 OF 30 PAGES

This document supersedes TM-2260/002/00

LISP II PROJECT

Memo No. 1

LISP II Internal Language

Abstract

This document describes the syntax and semantics of LISP II Internal Language.

INDEX

LISP II INTERNAL LANGUAGE

INTRODUCTION

The LISP II Internal Language (or IL) is a complete LISP-like language that serves three separate functions in LISP II:

- The semantics of LISP II are completely defined in terms of the IL.
- Source Language is defined in terms of its translation into IL. The compilation of LISP II programs is accomplished by translating source language into IL, then compiling and operating the resulting IL program. Macro expansion and saving of LISP II programs is performed in terms of IL.
- Programs can be input directly in IL, and the entire system can be operated completely in IL if desired, once the system has been informed properly.

The LISP II operating system is designed for on-line use. The executive program is called LISP and takes two arguments, which specify the input and output media. At entrance to the system, the function LISP (NIL, NIL) is called automatically.* The function LISP accepts a series of operations and performs them until the particular command STOP (); is encountered. STOP (); causes exit from the innermost LISP. The STOP (); command has no particular effect unless the LISP function has been called explicitly by the user, since after receiving a STOP (); at the outermost level, the system calls LISP (NIL, NIL) again.

*The arguments (NIL, NIL) mean that the standard teletype file (i.e., the one on which the user is logged in) is to be used. The values of these parameters in general are quoted names of files corresponding to such input/output devices as teletypes, disc, and magnetic tape.

The term top-level as used in this document always refers to the series of operations given to the LISP function. The semantics of the IL as given here applies either to operations input to the system in IL after the system has been so informed by the operation:

IL( ); in source language or

(IL) in internal language

or else applies to the stream of IL generated by the Syntax Translator from input in the Source Language form. However, since IL permits a wider range of expressions than any actual Syntax Translator will produce, the description of IL applies more completely to a stream of operations input directly in IL.

1. DATA

LISP II data types are an open ended set of things called datum. The first implementation will consist of

```
datum      =  constant
              quoted-expression

constant   =  Boolean
              number
              array
              function-specifier
              string

Boolean    =  TRUE
              false

false      =  FALSE
              NIL
              ( )

number     =  octal
              integer
              real

array      =  real-array
              integer-array
              symbol-array
              formal-array
              Boolean-array
              octal-array

atom       =  constant
              identifier

S-expression= atom
              (S-expression S-expression* {. S-expression|empty})

quoted-expression = (QUOTE S-expression)
```

Semantics

A constant has a particular representation in the computer, and an external input/output representation in the LISP II character set. In some cases, there may be several different input representations for the same constant. If so, the output representation is arbitrary but definite.

For example:

FALSE
NIL
( )

all represent the same constant. As a Boolean, it will print as FALSE. As a symbol, it will print as ( ). On the other hand, NIL can be input and means the same constant. Similarly, .0003 and 3.E-4 represent the same numerical constant, which will print out in a standard way, probably as 3.0E-4.

A quoted expression is a representation of a list structure similar to LISP 1.5 list structure, except for the existence of a wider spectrum of atoms. The printed representation of the value of a quoted expression (QUOTE s) is s.

The syntax of tokens and representation of constants for the Q-32 implementation of LISP II is given in LISP II Memo #11, TM-2260/004/00 entitled "The Syntax of Tokens."

2. TOP LEVEL OPERATIONS

The LISP IL is written as a series of operations in S-expression format.

```
operation    = declarative
               expression

declarative = section-declaration
              fluid-declaration
              function definition
              dummy-function-declaration
              macro-definition
              instructions-definition
              LAP-definition
```

Of the operations input at the top level, expressions constitute commands to the system to evaluate the expression and print out the resulting value (if any). Declaratives are simply absorbed by the system with some degree of error-checking being performed; thus a section declaration is simply accepted; a fluid-declaration or a dummy-functional declaration must be checked for inconsistency and be absorbed if correct; a function, macro-, or instruction-definition must be checked for syntax and consistency and then must be compiled. A definition to be compiled consists of an expression plus some declaration information. This section describes declarations made at the section level. The subjects of expressions and their evaluation are covered in sections 3 and 4.

2.1 THE SECTION-DECLARATION

```
section-declaration  =  (SECTION section-name type-option)

section-name   =  identifier

type-option    =  type
                  empty

type           =  simple-type
                  array-type
                  formal-type

simple-type    =  BOOLEAN|INTEGER|OCTAL|REAL|SYMBOL

array-type     =  (ARRAY f-type)

f-type         =  FORMAL
                  simple-type

formal-type    =  (FORMAL value-type indef-par-type parameter-type*)

value-type     =  NOVALUE
                  f-type

indef-par-type = (f-type transmission-mode INDEF)
                 empty
```

```
transmission-mode  =  LOC
                      empty

parameter-type  =  f-type
                   (f-type transmission-mode)
```

Semantics

The section declaration can be done only at top level of LISP. SECTION sets fluid variables in LISP which LISP has initialized to (SECTION NIL SYMBOL). A new section declaration replaces the old, and at exit from the function LISP, the previous section declaration is restored.

The use of an identifier as a section-name cannot conflict with any other uses of that identifier.

The section-name is used in compilation of all functions and in establishing all fluid bindings within the section. Bindings established within the NIL section are visible throughout other sections without tailing, unless there is a conflict with a binding made within the section. Bindings established within a named section are visible only within that section, or when tailed.

The type-option is a default declaration for all functions and fluid-variable declarations. Empty type-option implies SYMBOL by default. Example of the scope of section declarations is given in Fig. 1.

The type information contained in f-type and used in parameter-type, formal-type, array-type and value-type is a collapsed form of the more specific information contained in type. For every occurrence of array-type in type, SYMBOL is used in f-type. For every occurrence of formal-type in type, FORMAL is used in f-type. The complete specification of type occurs only in section declarations and in actual variable declarations. The abbreviated form f-type is used in dummy-function-declarations, value-type, and as sub-type information inside of array-type and formal-type.

2.2 SECTION-LEVEL BINDINGS

All of the following declarations, made at section level, establish bindings for identifiers, denoted in the syntax equations by f-name, and for variables, which can be tailed identifiers.

```
f-name  =  identifier

variable  =  f-name
             (EXTERNAL f-name)
             (EXTERNAL f-name section-name)
```

________ ________ ________	this is section ( ) of default-type SYMBOL
(SECTION NIL REAL) ________ ________ ________	still section ( ) but default type is REAL
(SECTION AA INTEGER) ________ ________	section AA, default-type INTEGER
(SECTION BB SYMBOL) ________ ________	section BB, default-type SYMBOL
(SECTION AA SYMBOL) ________ ________	back in section AA, but default-type is SYMBOL
(LISP input output) ________ ________	section ( ), default-type SYMBOL
(SECTION AA REAL) ________	section AA, default-type REAL
(STOP) ________	return to section AA, default-type SYMBOL

Fig. 1 Sections and default-types

A simple f-name variable refers to a variable declared in the current section, or in section NIL, with the current section declaration taking precedence. The form (EXTERNAL f-name) always refers to a variable declared in section NIL. The form (EXTERNAL f-name section-name) refers to a variable in the named section.

The tailed form of a variable may be used to establish variable declarations in any section, as well as to obtain the binding of the variable.

A variable must belong to one of the following mutually-exclusive classes:

- reserved identifier : Words such as FUNCTION, IF, DECLARE, SECTION, REAL, BLOCK, etc., are permanently reserved for system functions, and can never be declared in any other way, except that they may have property lists.

- fluid variable : Once a variable is declared to be fluid, the declaration cannot be changed without recompiling the entire section. The variable may be used as the name of a fluid parameter within a function declaration, but the fluid parameter must agree in type and mode with the prior fluid-declaration.

- function name : A variable used as a function name within a section can have only a single function declaration. There may be many dummy declarations for the same function name, but all must agree and must agree with the actual function declaration. A function may be excised and then redefined, but the new definition must agree in formal type with the old if the function is used by any other function in the section.

- macro or instructions name : Macros and instructions must be declared before use.

- unbound variables : Variables which are not yet declared at the section level in one of the above classes constitute a pool of available names which can be used at section level to name fluid variables, functions, macros and instructions.

2.3 FLUID-DECLARATION

fluid-declaration = (DECLARE {fluid-variable-declaration*)

fluid-variable-declaration = variable
(variable type-option storage-mode transmission-mode)

```
transmission-mode  =  LOC
                      empty
```

Semantics

A fluid-variable-declaration is required for all variables used free within a section, except for those which have been declared in section NIL, where the outer declaration is to hold within the named section.

As a fluid-variable-declaration, a variable alone means that the default-type applies, and all free use of the variable mean a fluid-variable of that type. A declaration of the form (variable type) is the same, except that a specific type has been declared for the fluid variable.

Storage-mode of FLUID means that all uses of this variable are fluid, namely that the current binding of the variable can be seen if the variable is used free (unbound) within an expression. If the variable is bound by a function in which the storage mode is not stated or it is declared FLUID, then the fluid storage mechanism applies to that variable. On entrance into the function, the previous value of the variable is stored and the new binding takes effect. On exit, the old binding is restored.

The transmission-mode LOC in a fluid-variable-declaration means that this variable is never used to hold a value directly, but instead always holds a locative pointer to a value of the specified type.

2.4 FUNCTION-DEFINITION

```
function-definition  =  (FUNCTION {variable|{(variable value-type)}}
                             p-list expression)

p-list  =  (indef-param param*)

indef-param  =  (p-name type-option storage-mode transmission-mode
                    INDEF p-name)|empty

p-name  =  variable

param  =  p-name
          (p-name type-option storage-mode transmission-mode)
```

Semantics

A function-definition in which type is not specified assumes the default-type of the section. All functions have an expression as a body.

In general, the value of the expression, converted to the proper type, is the value of the function. In NOVALUE functions, the value of the expression is not used.

The transmission-mode LOC means that this variable is to be transmitted by location rather than value (see section 3.5).

The parameter-storage-mode designation FLUID in a variable used as a parameter to a function affects the method of binding of that variable used in the function. If no FLUID mode has been designated at the section level and none is given in the function definition, the variable is strictly local and its binding cannot be referenced outside of the function itself. A FLUID declaration at the section level has the same effect as a FLUID declaration at the function-definition level. If a fluid-declaration is made at both the section level and the function definition, the type and transmission-mode declarations must agree. (See section 4.3, except that OWN storage mode is not possible for parameter-storage-mode.)

2.5 DUMMY-FUNCTION-DECLARATIONS

dummy-function-declaration = (FUNCTION (variable value-type indef-par-type parameter-type*)

A dummy-function-declaration provides information to the compiler sufficient to set up the calling sequence and value conversion. The actual function-definition must be consistent with all dummy-function-declarations.

Dummy-function-declarations contain transmission-mode information but do not contain storage-mode information. The correspondence between the type information in a dummy-function-declaration and the actual function declaration is given in section 2.1.

2.6 MACRO-DEFINITION

macro-definition = (MACRO variable (p-name) expression)

A macro-definition behaves like a function-definition of type SYMBOL and with one argument of type SYMBOL. A macro is a function which is applied by the compiler to the IL string before compilation.

Macros must be defined before use. Consequently, macros cannot be recursive, although a macro may be defined using a subsidiary, recursive function.

2.7 INSTRUCTIONS-DEFINITION

instructions-definition = (INSTRUCTION S (variable NOVALUE) ( ) expression)

An instructions-definition generates LAP code for the function it defines. The expression is intimately associated with the compiler, and makes use of the fluid variables and functions of the compiler. (See document on LISP II Compiler (to be published).)

2.8 LAP-DEFINITION

LAP-definition = (LAP listing d-list section-name)

listing = (desc-type f-name p-list item*)

desc-type = FUNCTION
MACRO
INSTRUCTIONS

item is as defined in the LAP II memo.

LAP and its use is described in LISP II Memo #10. A LAP-definition may be used to define a function, macro or instructions, depending upon the value of desc-type.

3. EXPRESSIONS

Expressions are the basic building block of LISP II. Syntactically, LISP II IL is written as a series of S-expressions, defined in section 1. An expression is the basic semantic unit of the language, and is one of a restricted set of S-expressions. Unlike declaratives, which are used at the top level, expressions are consistent at all levels of the LISP II language.

expression = simple-expression
conditional-expression
block-expression

simple-expression = datum
variable
form

This section will describe only simple-expressions and conditional-expressions. Block-expressions are described in section 1.

Datum was covered in section 1. A datum represents a constant or quoted expression. The value of a datum is the constant or quoted expression it represents.

The value of a variable is the binding of that variable at the level at which the evaluation takes place. Binding of variables at the top level is accomplished by declaring the variable FLUID and then using an assignment expression or evaluating an expression in which the variable is used free and set.

Syntactically,

```
form      =  (form-name argument*)
form-name =  variable
argument  =  expression
             functional
```

Semantically, the value and effect of a form depends upon the form-name.

```
form-name  =  array-variable
              function-name
              macro-name
              instruction-name
              formal-variable
```

These are semantic distinctions only and depend upon prior history, definitions and local context.

The following description of semantics of forms will cover assignment expressions, locatives, conditional and Boolean expressions, general evaluation of forms, and functional arguments.

3.1 ASSIGNMENT-EXPRESSION, LOCATIVES

```
assignment-expression  =  (SET locative expression)

locative       =  word-locative
                  list-locative

word-locative  =  full-locative
                  (BIT subscript subscript word-locative)
                  (BYTE subscript subscript expression)

list-locative  =  (PROP expression)
                  (CAR expression)
                  (CDR expression)

                  (LOC-ASSIGNMENT-EXPRESSION)
full-locative  =  variable
                  (array-name subscript subscript*)
LOC-ASSIGNMENT-EXPRESSION = (LOCSET LOC-VARIABLE FULL-LOCATIVE)
```

The value of an assignment-expression is that of the expression contained within.

An assignment-expression has the crucial side-effect of planting the value of the expression into the location specified by the locative, after making any necessary conversions, provided that the transformation is possible.

LOC-VARIABLE = variable

A full-locative translates into the address of a full word of memory. If the variable or array-name is not of type SYMBOL, then the address contains a value directly. In this case the assignment-expression places the value of the expression directly into the address. If the variable or array-name is of type SYMBOL, then a pointer to the value of the expression is placed into the locative address.

A word-locative having BIT modifiers means in general that only a portion of a word is to be set. If the variable or array-name is not of type SYMBOL, BIT specifies a portion of the word at the locative address. If the variable or array-name is of type SYMBOL, the BIT modifier is not permitted.

left — apparently not left ?

The first subscript in BIT specifies the right-most starting bit starting with Ø. The second subscript specifies the number of bits. Nested BIT modifiers are applied sequentially from inside out, the outer working on the portion remaining after the inner has had effect. by example

Thus:

(BIT 2 5 (BIT 1Ø 8 a)) = (BIT 12 5 a)

BYTE-modified word-locatives are defined only when the expression modified by that BYTE is a full-locative that points to a constant, or is a SYMBOL expression that points to a string. In the first case mentioned, BYTE works just like BIT, except that (BYTE 3 2 exp) is equivalent to (BIT 3n 2n exp) where n = number of bits per byte.

In the second case, BYTE finds or sets the appropriate number of characters in the string pointed to by exp, and its value is the selected number of bytes, left-justified into a word. Note that even though BYTE can find or set bytes in a string which occupies more than one word, it cannot set more than one word of data at one time, because its value must fit into one word.

The first subscript of BYTE specifies the left-most byte of a string. The second subscript specifies the number of bytes.

When used in an expression rather than a locative, the value of a BIT or BYTE modified expression is the right justified result of the bit or byte masking of the expression to which it is applied.

Thus, assuming a 48-bit word, with A initially zero

(SET (BIT 2 2 A) $\underbrace{\text{(BIT 2 2 15)}}_{3}$)

would set A to 12 and yield the value 3.

List-locatives work on SYMBOL type variables and manipulate list structure. Within a list-locative, the expression must produce a value of type SYMBOL. If (CAR X) is defined then (SET (CAR X) B) replaces the pointer (CAR X) by a pointer to the value of B. Similar results apply for CDR and the general $C\{A|D\}^*R$ functions.

The expression given as an argument to PROP must evaluate to an identifier. The value of (PROP expression) is the property list of the identifier. As a locative, PROP may be used to set the property list.

3.2 CONDITIONAL AND BOOLEAN-EXPRESSIONS

Conditional and Boolean expressions are special forms having a unique method of evaluation.

conditional-expression = (IF predicate expression {predicate expression}*
{expression|empty})

predicate = expression

A predicate is an expression which is subject to Boolean evaluation. The value of a predicate is FALSE if the expression it contains evaluates to FALSE or the empty list ( ), and is equivalent to TRUE otherwise.

In evaluating the conditional-expression (IF p_1 e_1 p_2 e_2 ... p_n e_n e_o), the predicates p_i are evaluated in turn from left to right, until one, say, p_j, is found that is TRUE (not FALSE). The value of the conditional expression is the value of the corresponding expression e_j. If none are true, then the value is e_o. If e_o is absent, and no predicate is true, the result will be a run-time error.

Except for any side effects that may occur in the evaluation of the p_i, the entire conditional-expression has the same effect as if it were replaced by the single e_j or e_o which is its value.

Boolean expression = (AND predicate*)
(OR predicate*)

(AND p_1 p_2 ... p_n) is TRUE if all p_i are TRUE (i.e., not FALSE) 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 FALSE, the remaining p_j for $j > i$ are not evaluated. (AND) is TRUE.

(OR p_1 p_2 ... p_n) is FALSE if all p_i are 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 TRUE, the remaining p_j for $j > i$ are not evaluated. (OR) is FALSE.

3.3 EVALUATION OF FORMS

For normal forms (function-name arg*), where all of the arguments are expressions, the evaluation of the form is done by evaluating all arguments, then passing the arguments to the function and operating the function.

The order of evaluation of arguments is not guaranteed. If it is desired to evaluate the arguments of form (f a b c d) in order, the block mechanism can be used, viz.,

```
(BLOCK ((A a) (B b) (C c) (D d)) (RETURN (f A B C D)))
```

3.4 FUNCTIONAL ARGUMENTS

```
functional        =  (FUNCTION {NIL|variable|({variable|empty} value-type)}
                               p-list expression funarg-variables)

                     formal-expression
                     (FUNCTIONAL formal-expression funarg-variables)

formal-expression =  function-name
                     expression

funarg-variables  =  (variable variable*)
                     empty
```

Semantics

A functional is a formal valued expression used as the argument of a function which requires a formal-type parameter, or to set or preset a formal variable or a variable of type SYMBOL.

The first format shown above creates a local function definition. The functional need have no name (i.e., can be of form (FUNCTION NIL ...) if it is not recursive. If the functional is used in setting a formal variable, presetting a formal variable, or as a formal argument of a function, there need not be any type information given in the functional, since the full type information is available to the compiler.

Any applicable FLUID storage mode information for parameters must be supplied, however.

The default type information for a functional is derived from the formal parameter in which it is used. For example, given

```
(FUNCTION (FF SYMBOL) SYMBOL (FORMAL INTEGER REAL (REAL LOC)))
```

if FF is called with

```
(FF A (FUNCTION B (X Y)   ... ))
```

then the functional B has value-type INTEGER and parameter types (X REAL) and (Y REAL LOC).

If the functional is used for setting a symbol type variable or a formal array, then full parameter type information is required.

Funarg-variables is an optional list of fluid variables. A variable is placed in the list if it is used free within the functional and if it is desired to save the binding of the fluid variable at the point at which the functional is bound and to use the saved value in evaluating any expression in which the formal variable is used, so that the functional binding is not affected by any intervening fluid bindings of the free variables. This is usually, but not always the desired interpretation for the free variable.

For example, consider

```
(FUNCTION (MAPCAR SYMBOL) ((X FLUID) (FN (FORMAL SYMBOL SYMBOL)))

     (IF (NULL X) NIL (CONS (FN (CAR X)) (MAPCAR (CDR X) FN)))

(FUNCTION (JX SYMBOL) (L (X FLUID))

          (MAPCAR L (FUNCTION ( ) (K) (CONS K X) (X))))

(JX (QUOTE (A B C D)) (QUOTE M))
```

Here, the use of the funarg-variable (X) was necessary in the definition of JX, to assure that the functional argument uses the value of X bound in JX, so that the result is ((A . M) (B . M) (C . M))

Without the funarg-variable declaration, the call to MAPCAR, as defined here with (X FLUID), would cause the binding of X in MAPCAR to be seen within the functional, and the result would be ((A A B C D) (B B C D) (C C D) (D D)) independent of the second argument of JX.

Although this example is artificial in that MAPCAR does not require (X FLUID), the principle applies to other cases of functional arguments.

3.5 FORMAL VARIABLES

A formal variable is a variable which has been declared formal so that it can receive a functional binding. A bound formal variable can be used in the same manner as a function-name. The formal-type declaration informs the compiler of the value-type and calling parameters of any functional which can be bound to the formal variable.

Once the formal-type has been declared, a formal variable can accept functional expressions of that type only.

In LISP II, unlike LISP 1.5, a functional expression cannot be applied to its arguments directly. Instead, the functional argument must first be set into a formal variable, and the formal variable then applied.

To operate a program at the top level of LISP II, one uses a formal variable and a functional expression where one would have used a LABEL LAMBDA expression and *FUNC in Q-32 LISP 1.5. For example:

```
(DECLARE (FF (FORMAL SYMBOL SYMBOL SYMBOL)))

(SET FF (FUNCTION ( ) (A B) (PLUS (TIMES A A) (TIMES B B))))

(FF 3 4)
```

would result in a printout of the value 25.

3.6 ARGUMENT TRANSMISSION

The arguments of a function are characterized by type and transmission mode. The expression that is used as the argument to a function must be consistent in type and mode with the argument declaration as follows:

1. Locative transmission:

   LOC. If the transmission mode LOC is specified for an argument of a function, then any expression used to supply the value of that argument must be a full-locative of the same type.

   For example:

   (FUNCTION (REALSET REAL) ((X LOC) Y) (SET X Y))

   is a function of two arguments (X REAL LOC) and (Y REAL) that sets the locative binding of X to the value of the expression Y.

   It is possible to call FN as follows:

   (REALSET A 3.5) (which sets A to 3.5), or

   (REALSET (AA i) 3.5) (which sets the i^{th} element of AA to 3.5),

   where A is a variable of type REAL and AA is a real array, but (REALSET 3.Ø 3.5) would be illegal and meaningless.

(In place of 3.5, any real-valued expression would suffice.)

In general, a variable must be declared LOC if the full-locative used as its argument is to be set as the variable itself is set.

A variable of array type must be declared LOC if the entire array is to be set by an assignment statement but not if only single cells in the array are to be changed. For example:

```
(FUNCTION (ARRAYSET SYMBOL) ((X (ARRAY REAL) LOC) (Y (ARRAY REAL)))
 (SET X Y))
```

which sets a real array variable X to a real array Y, must have a LOC declaration on X, since its result is to make the array variable specified by X point to an array Y.

However,

```
(FUNCTION (ARRAYSET1 SYMBOL)  (X (ARRAY REAL))
   (BLOCK ((M INTEGER))
       (FOR M (N STEP-1 UNTIL 1) (SET (X M) Y))
           (RETURN X)))
```

which sets N elements of the real array X to the value Y, does not require that X be LOC, since X will end up pointing to the same array at the end, but the values of the elements of the array will have been changed.

2. Arguments transmitted by value:

For arguments transmitted by value, any expression may be supplied in the function call, provided that the types are interconvertible.

The permitted conversions are shown in the following table:

TYPE FROM \ TO	B	I	O	R	S	a-t	f-t
BOOLEAN	X	-	-	-	S	-	-
INTEGER	TRUE	X	IO	IR	S	-	-
OCTAL	TRUE	OI	X	OR	S	-	-
REAL	TRUE	RI	RO	X	S	-	-
SYMBOL	P	SI	SO	SR	X	SA	SF
Array-type	TRUE	-	-	-	X	A	-
Formal-type	TRUE	-	-	-	S	-	F

Remarks:

X = exact, no conversion needed
\- = not permitted
S = symbol of appropriate type transmitted
TRUE = all non-Boolean values are TRUE
P = predicate evaluation: ( ) → FALSE, else TRUE
A = array-types must agree, else illegal
F = formal-types must agree, else illegal
IO = integer-to-octal conversion, exact, except -Ø → +Ø
IR = integer-to-real conversion, done by floating the integer
OI = octal-to-integer conversion, exact
OR = octal-to-real conversion, done by floating the equivalent integer
RI = real-to-integer conversion, rounded
RO = real-to-octal conversion, rounded
SI = if symbol is a number, convert to integer, else illegal
SO = if symbol is a number, convert to octal, else illegal
SR = if symbol is a number, convert to real, else illegal
SA = if symbol is an array and array types agree, transmit the value, else illegal
SF = if symbol is a formal-type and formal-types agree, transmit the formal, else illegal

3.7 LISP II ARITHMETIC

Arithmetic functions in LISP II IL consist of the primitive special forms PLUS, TIMES, MINUS, and DIFFERENCE which cannot be defined as functions, together with a set of primitive functions such as QUOTIENT, IQUOTIENT, REMAINDER, SIGN, etc., which are well-behaved functions.

In LISP II, arithmetic using PLUS, TIMES, MINUS, and DIFFERENCE is guaranteed to produce the same numeric values as if all arguments were of type symbol.

MINUS has one argument and produces a result of the same type as its argument, except that an octal input produces an INTEGER output. PLUS and TIMES take an indefinite number of arguments. DIFFERENCE takes two arguments.

The type of the results of PLUS, TIMES, and DIFFERENCE is related to the input type by the following table:

	INTEGER	OCTAL	REAL	SYMBOL-IO	SYMBOL-R
INTEGER	INTEGER	INTEGER	REAL	SYMBOL-I	SYMBOL-R
OCTAL	INTEGER	INTEGER	REAL	SYMBOL-I	SYMBOL-R
REAL	REAL	REAL	REAL	REAL ~~SYMBOL-R~~	REAL ~~SYMBOL-R~~
SYMBOL-IO	SYMBOL-I	SYMBOL-I	REAL ~~SYMBOL-R~~	SYMBOL-I	SYMBOL-R
SYMBOL-R	SYMBOL-R	SYMBOL-R	REAL ~~SYMBOL-R~~	SYMBOL-R	SYMBOL-R

In the table SYMBOL-IO means either SYMBOL INTEGER or SYMBOL OCTAL, SYMBOL-I means SYMBOL INTEGER, and SYMBOL-R means SYMBOL-REAL.

The output type of PLUS and TIMES can be obtained by successive applications of the table to the partial sums or products.

The order of combination of the arguments in PLUS and TIMES is not guaranteed.

The function QUOTIENT in LISP II has arguments and value of type REAL.

IQUOTIENT and REMAINDER have arguments and value of type INTEGER.

The predicates
(EQUAL x y) meaning is $X = Y$
(GR x y) meaning is $X > Y$
(LS x y) meaning is $X < Y$
(GQ x y) meaning is $X \geq Y$
(LQ x y) meaning is $X \leq Y$
(NQ x y) meaning is $X \neq Y$

are all exact. While EQUAL and NQ work on all types of arguments, the compiler compiles these predicates open and produces efficient code for them where possible.

4. BLOCK

```
block-expression     =  (BLOCK (block-declaration*) {label|statement}*)

block-declaration    =  switch-declaration
                        block-variable-declaration

label  =  identifier

statement  =  compound-statement
              block-statement
              go-statement
              conditional-statement
              return-statement
              code-statement
              simple-expression
              (LABEL label statement)

compound-statement  =  (BLOCK (switch-declaration*) {label|statement}*)

block-statement   =  (BLOCK block-stat-decls {label|statement}*)
                     for-statement
                     try-statement

block-stat-decls  =  (block-declaration* block-variable-declaration
                             block-declaration*)

block  =  block-statement
          block-expression
```

Semantics

A block-expression is a block or compound-statement used where an expression is called for, and in general evaluated to produce a value. Statements occur only inside of block-expressions.

A block-statement differs from a compound-statement only in that a block-statement must contain at least one block-variable-declaration, while a compound-statement can not contain any block-variable-declarations. Other forms of block-statements are form-statement, which is macro-expanded into a block-statement that may contain a block-variable-declaration (see section 4.6) and try-statement (see section 4.8).

4.1 BLOCK-VARIABLES

```
block-variable-declaration  =  variable
                               (variable type-option storage-mode)
                               (var-preset-declaration)

var-preset-declaration  =  (variable type-option {storage-mode | OWN | expression)
                           (variable ASSIGNED expression)
                           (variable type-option storage-mode LOC full-locative)
```

Semantics

Block variables, or variables declared at the block level, are initialized at entrance into the block. If type-option is empty, and the variable has not been declared FLUID at a higher level, then the type is the default-type of the function or section, as in the case of parameter declarations. If a section-level FLUID declaration is in effect for the variable, the type is determined by the previous declaration, and the block-level declaration must be consistent in type with the previous declaration.

Initialization of FLUID variables causes fluid binding to occur; namely, the old value of the fluid variable is stored on the pushdown list. When the block is exited in any manner, the bindings of all FLUID variables are restored to the previously stored values.

A variable declared with OWN is a fluid variable but is used free within the block, and is neither fluid-bound at entrance to the block, nor restored at exit.

Except for OWN variables, all variables that are declared at block level are preset upon entrance to the block. If a var-preset-declaration is given, the preset value is the value of the expression given in the declaration. Variables whose transmission-mode is LOC must be preset to a full-locative.

An OWN variable declaration must contain a preset expression; however, the OWN variable is preset only if the variable has not previously been set.

If no preset information is given, a variable is set to NIL or zero at the entrance to the block.

The form (variable ASSIGNED expression) implies both a type and a preset. The variable, which must be local, is set to the same type as the value of the expression used to preset it.

Local variables, (i.e., those not FLUID or OWN) are visible only within the block in which they are declared and within all inner blocks in which they are used free. They cannot be used in functional arguments, and cannot conflict with any other local or fluid variables of the same name.

4.2 GO-STATEMENT, LABEL, AND SWITCH

```
go-statement  =  (GO label)
                 switch-call

switch-declaration  =  (switchname SWITCH s-label*)

switchname  =  identifier

s-label  =  label
            NIL

switch-call  =  (GO (switchname subscript))
```

<u>Semantics</u>

A label or switchname must be unique within the single functional or within the single top-level expression or definition in which it resides. The use of an identifier as a <u>label</u> or <u>switchname</u> cannot conflict with any other use of that identifier.

A label is regarded as a symbolic name for the first statement that follows it, and is used to transfer control to that statement. A label located after the last statement in a block or compound-statement is used to cause control to "fall through."

The scope of a label consists of all statements contained within the innermost block in which the label occurs, but excluding all expressions contained within the block. It is possible to "go to" a label (i.e., (GO label) is legal) from anywhere within the scope of the label.

A switch-declaration can contain a label only if it lies within the scope of that label, or the s-label can be NIL. The scope of a switch is the same as that of a label at the top level of the block or compound-expression.

Apart from binding of variables, the evaluation of a block or compound-statement consists of operating each statement in turn, until either the control "falls through" after the final statement in the block or compound-statement, or until a go-statement, return-statement, or an exit-expression is encountered.

If the control "falls through" in a block-expression, the value of the block-expression is NIL. If the control "falls through" a block-statement or compound-statement control passes to the next statement outside of that block-statement or compound-statement.

A go-statement encountered within a block or compound-statement causes control to be transferred to the label contained in the go-statement. If the label lies outside of a block-statement, a block-exit is performed before the control is actually transferred. The scope definition for label permits "going out of" a block but prohibits "going into" a block.

A switch-call causes a transfer of control to one of the labels in a switch-declaration, or "falls through" depending upon the switch-declaration and value of its subscript. The s-labels on a switch-declaration can be NIL or they can be any labels in whose scope the switch-declaration occurs.

When a switch call is encountered, the subscript expression is evaluated to yield an integer, and the integer is used to select one of the s-labels in the switch. The s-labels in the switch declaration correspond to subscript values 1, 2, ..., n. If an s-label exists for the particular value of the subscript, then the effect of the switch call is the same as (GO s-label). If no s-label exists, i.e., if subscript < 1 or subscript > n or if the corresponding s-label is NIL, then the switch call is not defined.

4.3 CONDITIONAL-STATEMENT

conditional-statement = (IF predicate statement {predicate statement}*
{statement|empty})

Semantics

A conditional-statement is evaluated by evaluating the predicates from left to right until the first TRUE (non-NIL) predicate is found. If one is found, the following statement is operated. If all predicates are FALSE, the final statement is operated, or if there is no final statement, control "falls through" to the next dynamic statement outside of the conditional statement.

Any top-level statement inside of a conditional-statement may be labelled by the form (LABEL label statement). Such a label is visible at the same level as that of the conditional-statement itself. If control is transferred into a conditional-statement by (GO label), the statement immediately following the label is operated, and (if it was not a go-statement or a return-statement) control "falls through" to the next dynamic statement outside of the conditional-statement.

4.4 RETURN-STATEMENT

return-statement = (RETURN expression)

Semantics

The hierarchy of statements in LISP II assures that every return-statement lies inside of a block-expression (i.e., one which is being used and evaluated as an expression).

Whenever a return-statement (RETURN expression) is encountered in the flow of control within a block or compound-expression, the effect is the following:

1. The expression is evaluated.
2. Exit is made from all compound-statements and block-statements in which this return-statement occurs, with restoration of fluid variables occurring at each level, until the block-expression is reached.
3. The value of the evaluated expression, appropriately converted to the proper value type, is the value of the block-expression.

4.5 CODE-STATEMENT

code-statement = (CODE item*)

item = label
instruction
pseudo-instruction

Semantics

Instructions and pseudo instructions and the use of code-statements are defined in the LAP II memorandum.

Code-statement are used to enter machine coded instructions into a program. The labels that occur within code-statements are visible at the same level as the code-statement itself.

4.6 FOR-STATEMENT

for-statement = (FOR variable for-element for-element* statement)

does it for last case / i.e test done after

for-element = expression
(a-expr STEP a-expr {term-element|UNTIL a-expr})
(expression {RESET expression|empty} term-element
({IN|ON} expression term-element)

term-element = WHILE predicate
UNLESS predicate
empty

a-expr = expression

An a-expr is an expression whose value is numeric.

Semantics

1. A for-statement is a statement, not an expression. The variable in the for-statement can be any variable bound at a higher level. The statement which forms the body of the for-statement may be any statement, including another for-statement. If, at any iteration, a statement to be executed as the body of the for-statement collapses into a go-statement or return-statement, it causes an unconditional exit from the for-statement.

2. A single for-statement with more than one for-element is exactly equivalent to a sequence of primitive for-statements having the same variable and statement body, e.g.,

   (FOR v f_1 f_2 f_3 ... f_n s)

   where v is a variable, f_1, f_2 ... f_n are for-elements,

   and s is a statement, is precisely equivalent to the sequence of for-statements:

   (FOR v f_1 s) (FOR v f_2 s) ... (FOR v f_n s)

   The semantics of any for-statement can therefore be described in terms of the primitive for-statement (or p.f.s.)

   (FOR v f s)

   which depends upon the for-element f as follows:

(BLOCK ( )

3. If f is an expression, then the p.f.s. is equivalent to (SET v f) s)

4. If f = (a_1 STEP a_2 UNTIL a_3),
where a_1, a_2, and a_3 are a-expr, then the p.f.s. is equivalent to:

(BLOCK ((g ASSIGNED a_1)) (SET v g) ℓ_1 s (SET g a_2)
  (IF (GR (TIMES (SIGN G) (DIFFERENCE (PLUS v g) a_3)) Ø)
    (GO ℓ_2)) (SET v (PLUS v g) (GO ℓ_1) ℓ_2)

(BLOCK ( )

5. If f = (e_1 {STEP a_2|RESET e_2|empty} {WHILE p|UNLESS p|empty}),)
the f.p.s. is equivalent to:

(SET v e_1) ℓ_1 s {(IF {NOT p)|p} (GO ℓ_2))|empty}
  {(SET v (PLUS v a_2))|(SET v e_2)|empty} (GO ℓ_1) ℓ_2

where ℓ_1 and ℓ_2 are generated labels and (NOT p) corresponds to WHILE.

6. If f = ({IN|ON} e_1 {WHILE p|UNLESS p|empty}),
the p.f.s. is equivalent to

(BLOCK ((g SYMBOL e_1))
  (SET v {(CAR e_1)|e_1}) ℓ_1 s (IF (NULL (SET v (CDR v))) (GO ℓ_2)
    {(NOT p)|p|empty} (GO ℓ_2)) (GO ℓ_1) ℓ_2)

Where ℓ_1, ℓ_2 and g are generated identifiers, and IN corresponds to (CAR g), ON to g and the three choices in the conditional statement correspond to the WHILE/UNLESS/empty cases.

The compiler will actually implement most forms of for-statement by means of macro expansion similar to that indicated here.

4.7 SIMPLE EXPRESSION USED AS A STATEMENT

Any expression can be used as a statement. The expression used in this way is evaluated and the value discarded. Thus this form of statement is useful only if it produces side effects, such as setting variables and performing input-output functions.

(Syntactically, only simple-expression is included in the definition of statement, since compound-expression and conditional-expressions are already subsumed as special cases of compound-statements and conditional-statements.)

4.8 TRY-STATEMENT AND EXIT-EXPRESSION

try-statement = (TRY statement full-locative statement)

exit-expression = (EXIT expression)

Semantics

A try-statement is a block containing two statements and a full-locative.

The first statement is executed normally unless an exit-expression is encountered within it. If no exit is encountered, the second statement is bypassed, and if the first statement "falls through," the try-statement "falls through."

If an exit-expression is encountered, control reverts to the innermost try-statement in which the exit-expression occurs, and the effect is the same as

(SET full-locative expression)

statement, where full-locative and statement are those given in the try-statement, and the expression used is that given in the exit-expression.

The full-locative used in the try-statement should be of type SYMBOL, so that it can accept the value of the expression.