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

and subcontract 65-107.

TECH MEMO

a working paper

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

Information International Inc. / 200 Sixth Street / Cambridge, Massachusetts 02142

TM-2710/220/00

AUTHOR S. L. Kameny

L. Hawkinson, I.I.I.

TECHNICAL M. Levin, I.I.I.

S. L. Kameny, S.D.C.

RELEASE A. L. Fenaughty, I.I.I.

S. L. Kameny, S.D.C.

for D. L. Drukey

DATE 4 November 1965 PAGE 1 OF 39 PAGES

LISP II INTERMEDIATE LANGUAGE

ABSTRACT

This document describes the syntax and semantics of the LISP II Intermediate Language, and constitutes the basic definition of LISP II. All LISP II source language programs are converted into the Intermediate Language before compilation. The Intermediate Language can be input directly to LISP II.

FOREWORD

LISP II is a joint development of SDC and III. The idea for LISP II as a language combining the properties of an algebraic language like ALGOL and the list-processing language LISP was conceived by M. Levin of MIT. Development of the concepts of LISP II was carried forth in a series of conferences held at MIT and Stanford University. Contributions in concepts and detail were made by Prof. John McCarthy of Stanford University, Prof. Marvin Minsky of MIT, and the LISP II project team consisting of M. Levin, L. Hawkinson, R. Saunders and P. Abrahams of III, and S. Kameny, C. Weissman, E. Book, Donna Firth, J. Barnett and V. Schorre of SDC.

For the implementation of LISP II, it was decided to define a standard, computer-independent, LISP-like intermediate language and to define the LISP II source language in terms of its translation into the Intermediate Language.

This document describes the syntax and semantics of the Intermediate Language.

CONTENTS

LISP II INTERMEDIATE LANGUAGE

1. INTRODUCTION

The LISP II Intermediate 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 is called automatically in the form (LISP NIL NIL).* 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 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

---

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

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.

2. CONSTANTS

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

```
constant   =  simple-datum
              quoted-expression
              function-specifier

simple-datum  =  Boolean
                 number
                 array
                 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   =  simple-datum
          identifier

datum  =  S-expression

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

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

Semantics

A datum 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 datum. If so, the output representation is arbitrary but consistent.

For example:

```
FALSE
NIL
( )
```

all represent the same datum. 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 datum. Similarly, 0.0003 and 3.E-4 represent the same numerical datum, which will print out in a standard way, probably as 3.0E-4.

A quoted-expression represents a datum in Expression context. It is like QUOTE in LISP 1.5 , 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 TM-2710/210/00 entitled "The Syntax of Tokens."

3. TOP LEVEL-OPERATIONS

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

```
operation   = declarative
              expression

declarative = section-declaration
              free-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 free-declaration or a dummy-function - declaration must be checked for consistency and be absorbed if correct; a function, macro-, or instructions-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 evaluations are covered in sections 4 and 5.

3.1 THE SECTION-DECLARATION

```
section-declaration = (SECTION {section-name | (section-name*)} type-option)

section-name     =  identifier
                    NIL

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. Whenever the function LISP is called, the section-name and section-type are initialized to NIL and SYMBOL, respectively, unless an explicit section-declaration is encountered. 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.

If the section-declaration contains only the single section-name NIL, then the current section is NIL and there are no default sections. If the section-declaration has a single section-name other than NIL, the section-name constitutes the current section, and NIL is the default section.

If the section-declaration contains a list of section-names, the first name in the list is the current section, and the remaining names on the list are taken in succession as default sections, with section NIL assumed as the final default section if it does not appear earlier on the list.

The current section is an implicit parameter of section-level declarations. The current section and default sections are used to determine the declarations for free-variables, as described below.

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.

3.2 SECTION-LEVEL DECLARATIONS

Declarations, made at section level, establish type and transmission mode for variables.

```
variable  =  untailed-variable
             tailed-variable

untailed-variable  =  f-name

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

f-name  =  identifier
```

______________ ______________ ______________	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

Figure 1 Sections and default-types

A simple untailed-variable declared at section level refers to a variable declared in the current section. The tailed form (EXTERNAL f-name) always refers to a variable declared in section NIL. The tailed form (EXTERNAL f-name section-name) refers to a variable in the named section. Thus, through use of the tailed forms, variable declarations can be established in sections other than the current one.

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

fluid-variable
own-variable
function-name
macro-name
instructions-name

Once the declaration for a variable is in effect, a subsequent declaration for the same variable is permitted only if it agrees in type with the previous declaration, or if there are no references to that variable from assembled code or synonyms.

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, own variables, functions, macros and instructions.

3.3 FREE-DECLARATION

```
free-declaration  =  (DECLARE free-variable-declaration*)

free-variable-declaration  =  variable
                              (variable type-option free-storage-mode
                                 {LOC|empty})
                              free-var-preset-decl
                              synonym-declaration

free-var-preset-decl  =  (variable type-option free-storage-mode expression)
                         (variable type-option free-storage-mode LOC
                            full-locative)

free-storage-mode  =  OWN
                      storage-mode

storage-mode  =  FLUID
                 empty

synonym-declaration  =  (variable MEANS variable)
```

Semantics

free-variables:

A free-variable-declaration is required for all variables used free; a variable is used free if it is referred to within a function or functional expression, or used in an expression at section level, without being bound as a block variable or as an argument of the function or functional expression.

For any untailed variable used free within a section, the declarations of the current section take precedence in determining type and storage mode. If more than one declaration has been made for this variable in the current section, the last declaration applies. The type of the variable is that specified in the free-variable-declaration, except that if no type is given in the free-variable-declaration, the section-type is used as a default.

If the variable has no free-variable-declaration in the current section, but has been declared in a default section, the variable acts as a tailed-variable belonging to the section in which it was declared. (If the variable has been declared in more than one default section, then the declaration used is that in the section whose section-name occurs first on the section-list.)

A tailed-variable belongs to the section named by its tailing, and must have a declaration in that section.

fluid-variables:

Storage-mode FLUID means that all uses of this variable are fluid, and all bindings of the variable within a block or function are fluid, whether or not the variable is explicitly declared FLUID within the function or block. Storage-mode empty means that the section-level declaration (and this fluid binding) applies only in functions or blocks in which the variable is explicitly declared to be FLUID. In either case, all free uses of the variable refer to the most recent setting of the fluid variable.

fluid binding:

Fluid binding is applied to a fluid variable that is used as a block-variable or as a parameter of a function whenever that function or block is entered. Fluid binding of a variable is accomplished by first saving its previous value on the pushdown list and then setting it (the fluid variable) to the corresponding argument in the case of a function, or to the preset value in the case of a block. All uses or settings of the fluid variable within the function (or block) refer to or affect the value of the fluid variable. Exiting from the function or block in any manner causes the previous value to be obtained from the pushdown list and restored to the fluid variable.

OWN variables:

The free-storage mode OWN in a free-variable declaration means that this variable can be set and used as a free variable but can never be bound by a block or function. Variables declared as FUNCTION, MACRO, or INSTRUCTIONS behave as OWN when used free. (The use of a variable as an OWN, FUNCTION, MACRO, or INSTRUCTIONS variable does not conflict with the use of a lexical variable of the same name. See section 3.4.)

free-var-preset-decl:

A free variable preset declaration contains declaration information for a variable together with an expression or full-locative to be used to preset the variable. It is equivalent in effect to a free-variable-declaration made without a preset, followed by a setting of the variable to the value of the expression contained in the free-var-preset-decl.

locatives:

The transmission-mode LOC in a free-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. The expression used to preset a locative free-variable must be a full-locative, which is an expression that evaluates to a locative pointer (see section 4.6).

synonym-declaration:

The synonym-declaration (aa MEANS bb) means that variable aa is another name for variable bb. The synonym-declaration is legal only if a declaration already exists for bb. If a declaration is already in effect for aa, then aa and bb must agree in type. Synonym declarations are transitive and dynamic. The effect of the two synonym declarations

(bb MEANS cc)

(aa MEANS bb)

is the same as the effect of

(bb MEANS cc)

(aa MEANS cc)

In other words, if the variable on the right hand side of a synonym is itself a synonym, it is replaced by its meaning. Hence, the state of a variable at any time is reflected by a synonym relationship that is only "one deep."

A synonym-declaration with the same variable on both sides of MEANS, i.e.,

(aa MEANS aa)

is treated as a special case. If aa was a synonym, the synonym relationship is removed, and aa now means itself. If aa was not a synonym, nothing happens.

Synonym declarations are dynamic. The meaning of a variable depends upon the last operated synonym declaration for that variable.

3.4 FUNCTION-DEFINITION

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

par-list = (indef-param param*)

indef-param = (indef-name type-option transmission-mode (INDEF par-name)
empty

par-name = variable

param = par-name
(par-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.

Each parameter or param on the par-list represents an argument for the function being defined. The par-name is the name of the variable, and has only local or lexical significance unless storage-mode is FLUID or a FLUID free-variable-declaration exists for this variable. The type of a parameter is either specified in type-option, or if the type-option is empty, the parameter-type is obtained by default from the value-type of the function, or, if the function is NOVALUE, the section-type is used as the parameter type.

A function may have an indef-param as its first argument. An indef-param represents an indefinite number of arguments all of the same type. The indef-name in the indef-param is a variable used within the function, and must be a lexical variable. (Note that the indef-param does not contain storage-mode.) The par-name, following the word INDEF, is always a lexical variable of type INTEGER, and represents the number of arguments. When the function is called, the arguments supplied for the indef-param are stored sequentially on the push-down list, and the function receives as parameters a locative corresponding to the number of indef arguments and then all subsequent arguments. Within the function, the indef-name must always be subscripted to obtain the indef argument values.

In any parameter, the transmission-mode LOC means that this parameter is to be transmitted by location rather than value (see section 4.6). 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 value is not available outside of the function itself.

A local or lexical variable is simply a name for a temporary storage cell on the pushdown list, and the binding of an argument to a lexical variable used as a parameter of a function is accomplished by storing the argument on the pushdown list. The use of a variable as a lexical variable cannot conflict with its use as an OWN variable. A tailed-variable is never a lexical variable.

A FLUID declaration made at the function-definition level causes FLUID binding of that variable to occur at entry to the function, as discussed in section 4.4. A fluid-declaration of a variable in a function-definition establishes the type and transmission-mode for the variable, and is equivalent to a free-variable-declaration of the form (variable type).

3.5 DUMMY-FUNCTION-DECLARATIONS

dummy-function-declaration = (FUNCTION {variable|(variable value-type)}
{par-list|(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-definition is given in section 3.1.

3.6 MACRO-DEFINITION

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

A macro-definition behaves like a function-definition of type SYMBOL and with one argument of type SYMBOL. If M is a macro name, then wherever the form (M ...) is to be compiled, the value of the macro-definition of M, applied to the argument (M ...), is compiled in its place.

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

3.7 INSTRUCTIONS-DEFINITION

instructions-definition = (INSTRUCTIONS (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--TM-2710/320/00).

3.8 LAP-DEFINITION

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

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

desc-type = FUNCTION
MACRO
INSTRUCTIONS

item is as defined in the LAP II memo.

LAP and its use is described in TM-2710/250/00. A LAP-definition may be used to define a function, macro or instructions, depending upon the value of desc-type.

4. 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 2. 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 = constant
variable
form

This section will describe only simple-expressions and conditional-expressions. Block-expressions are described in section 2.

Constant was covered in section 2. A constant is a simple-datum or quoted-expression. The value of a constant is the datum it represents.

The value of a variable is the most recent setting of that variable at the level at which the evaluation takes place. Setting of variables at the top level is accomplished by declaring the variable FLUID or OWN and then using an assignment expression, by evaluating an expression in which the variable is used free and set, or by means of a free-variable-preset-declaration.

Syntactically,

form = (form-name argument*)

argument = expression
functional

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

form-name = array-variable
function-name
macro-name
instructions-name
formal-variable
indef-name

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.

4.1 ASSIGNMENT-EXPRESSION, LOCATIVES

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

locative = word-locative
list-locative

word-locative = full-locative
(CORE subscript)
(BIT subscript subscript word-locative)

subscript = expression

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

full-locative = variable
(array-name subscript subscript*)
loc-assignment-expression

loc-assignment-expression = (LOCSET loc-variable full-locative)

loc-variable = variable

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 conversion, where such conversion is possible.

A word-locative represents a designated portion (possibly all) of a word of memory. When a word-locative is used as the first argument of an assignment-expression, the assignment expression causes the converted and possibly truncated value of its second argument to replace the designated portion of that word.

A subscript is an arithmetic expression which is evaluated to produce an integer value. Subscripts are used to specify particular elements of an array or indef-param.

The word-locative (CORE subscript) refers to a particular location in core storage whose address is equal to the subscript value, and can be used to obtain data from or store into a particular core location. Its value is of type OCTAL.

The word-locative (BIT subscript subscript w) is used to designate a portion of a word w, a word-locative of type OCTAL.

The first subscript in BIT specifies the right-most 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.

Thus:

(BIT 2 5 (BIT 1Ø 8 w)) = (BIT 12 5 w)

When it is used as an expression rather than a locative, the value of the BIT modified expression is of type OCTAL and equal to the selected portion of word w, right justified.

List-locatives work on identifiers and list structure. The expression used as the argument of a list-locative must produce a value of type SYMBOL. If (CAR X) is defined then (SET (CAR X) B) replaces the symbol (CAR X) by the symbol 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.

A *loc-assignment-expression* is used to change the locative pointer in the *loc-variable* to the full-locative value of the second argument. A *loc-variable* is a variable which has a transmission mode of LOC. The value of the loc-assignment-expression is the full-locative, which must agree in type with the loc-variable.

4.2 CONDITIONAL AND BOOLEAN-EXPRESSIONS

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

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

logical-expression = expression

A logical-expression is an expression which is subject to Boolean evaluation. The value of a logical-expression is FALSE if it 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 logical expressions 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 logical-expression is true, the conditional expression is undefined and will cause 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 logical-expression*)
(OR logical-expression*)

(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.

4.3 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.

4.4 FUNCTIONAL ARGUMENTS

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

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

formal-expression  =  function-name
                      expression

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

Semantics

A functional is a formal-valued expression. A functional must be used as the argument of a function which requires a formal-type parameter, or to set or preset a formal variable; it may also be used wherever an arbitrary symbol is permitted.

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.

For example, given the dummy functional declaration

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

then in the form

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

the functional B has value-type INTEGER and params (X REAL FLUID) and (Y REAL LOC).

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

A formal-expression is any expression whose value is a formal. Any function-name is automatically a formal expression.

Funarg-variables is an optional list of fluid variables, which should be composed of variables that are used free within the functional. A variable is placed on the list if it is desired to save the binding of the variable at the point at which the functional is evaluated for later restoration when the formal (value of the functional) is applied to arguments. This assures that free use of the fluid within the functional will not be affected in any way by any possible rebinding of the variable occurring between the point of evaluation and point of application of the functional. **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.

4.5 FORMAL VARIABLES

A formal variable is a variable which has been declared formal so that it can receive a dynamic functional setting. After having been given a proper setting, a 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 parameter-types of any functional to which the formal variable can be set.

Once its type has been declared, a formal variable can accept only those functional bindings whose value-type and parameter-types match those of the formal.

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 or 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.

4.6 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:

Locative transmission:

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. An 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 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.) (which sets the ith element of AA to 3.5),

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

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))(Y 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.

Arguments transmitted by value:

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

Type conversion is discussed in section 4.7.

4.7 TYPE CONVERSION

Type conversion is required whenever the value type of an expression differs from the type expected at the point at which the value is used. The permitted conversions are described in Table 1.

Table I

Type Conversion

TYPE FROM \ TO	B	I	O	R	S	a-t	f-t
BOOLEAN	X	-	-	-	X	-	-
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	-	-	-	X	-	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 -∅ → +∅
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

If it is desired (for system work only) to suppress automatic type conversion, "cheater functions" can be employed. A cheater-function changes the apparent type of a value without actually converting the value. The available cheater functions are given in Table II.

Table II

Cheater Functions

Name	Argument Type	Apparent value type
B2O.	BOOLEAN	OCTAL
I2O.	INTEGER	OCTAL
R2O.	REAL	OCTAL
S2O.	SYMBOL	OCTAL
F2O.	FORMAL	OCTAL
O2B.	OCTAL	BOOLEAN
O2I.	OCTAL	INTEGER
O2R.	OCTAL	REAL
O2S.	OCTAL	SYMBOL
O2F.	OCTAL	FORMAL

By the use of two "cheater-functions" any f-type can be converted to any other apparent type. CAUTION!

4.8 LISP II ARITHMETIC

Arithmetic functions in LISP II IL consist of the primitive INSTRUCTIONS 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.

In LISP II, arithmetic using PLUS, TIMES, MINUS, and DIFFERENCE is guaranteed to produce the same numeric values when any or all arguments are real or integer, as they would 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 of two arguments is related to the types of its input arguments by the following table:

Table III

Results of PLUS, TIMES or DIFFERENCE of two Arguments

Argument \ Argument	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	REAL
SYMBOL-IO	SYMBOL-I	SYMBOL-I	REAL	SYMBOL-I	SYMBOL-R
SYMBOL-R	SYMBOL-R	SYMBOL-R	REAL	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 of more than two arguments 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 logical expressions

(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. The forms GR, LS, GQ, and LQ work on numeric arguments only, while EQUAL and NQ work on all types of arguments. The compiler compiles these logical-expressions open and produces efficient code for them where possible.

5. 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 for-statement, which is macro-expanded into a block-statement that may contain a block-variable-declaration (see section 5.6) and try-statement (see section 5.8).

5.1 BLOCK-VARIABLES

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

var-preset-declaration  =  (variable type-option storage-mode 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 the top level(by means of a free-variable-declaration), then the type is the default-type of the function or section, as in the case of parameter declarations. If a FLUID free-variable declaration is in effect for the variable, the type is determined by that declaration, and the block-level declaration must be consistent in type with the free-variable declaration.

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

A variable which has previously been declared OWN cannot be bound as a fluid variable.

All variables that are bound 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-code is LOC must be preset to a full-locative.

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

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

Lexical 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 fluid variables of the same name. When a variable is found free within an expression, the most recent setting (FLUID, OWN, or lexical) is used.

5.2 GO-STATEMENT, LABEL, AND SWITCH

go-statement = (GO label)
switch-call

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

switchname = identifier

s-label = label

switch-call = (GO (switchname subscript))

Semantics

A label or switchname must be unique within the single functional or within the single top-level expression or definition in which it resides. However, the use of an identifier as a label or switchname 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. The scope of a switch is the same as that of a label at the top level of its 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. The s-labels on a switch-declaration 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, then the switch call is not defined.

5.3 CONDITIONAL-STATEMENT

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

Semantics

A conditional-statement is evaluated by evaluating the logical-expression from left to right until the first TRUE (non-NIL) predicate is found. If one is found, the following statement is operated. If all logical-expressions are FALSE, the final statement is operated, or if there is no final statement, nothing is operated.

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.

5.4 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.

5.5 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 document TM-2710/250/00.

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.

5.6 FOR-STATEMENT

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

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

```
term-element = WHILE logical expression
               UNLESS logical-expression
               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.

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:

   (BLOCK ( ) (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 shown in 3, 4, and 5.

3. If f = (expression), then the p.f.s. is equivalent to

   (BLOCK ( ) (SET v f) s)

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

```
(BLOCK ((g ASSIGNED v))
   {(SET v a1)|empty}
 ℓ1 {IF {(NOT p)|p} (GO ℓ2)}|empty}
   s
   (SET v (PLUS v (SET g a2)))
   (IF (LQ (TIMES (SIGN g) (DIFFERENCE v a3)) 0)
       (GO ℓ1))
ℓ2 )
```

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

```
(BLOCK ( )
   {(SET v e1)|empty}
 ℓ1 {(IF {(NOT p)|p} (GO ℓ2))|empty}
   s
   {(SET v (PLUS v a2))|(SET v e2)|empty}
    (GO ℓ1)
ℓ2    )
```

where ℓ_1 and ℓ_2 are generated labels and (NOT p) corresponds to WHILE. If none of the terms STEP, RESET, WHILE, or UNLESS are present in f, the statement (GO ℓ_1) is not generated.

6. If f = ({IN|ON} e_1 {WHILE p|UNLESS p|empty}),

the p.f.s. is equivalent to

```
(BLOCK ((g  SYMBOL e1))
  ℓ1 (IF (NULL g) (GO ℓ2))
     (SET v {(CAR g)|g})
     {(IF {(NOT p)|p} (GO ℓ2))|empty}
     s
     (SET g (CDR g))
     (GO ℓ1)
ℓ2   )
```

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 below.

5.7 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.)

5.8 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 that of operating the block.

```
(BLOCK ( )

(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.

INDEX TO SYNTAX EQUATIONS

INDEX TO SYNTAX EQUATIONS (Cont'd.)

INDEX TO SYNTAX EQUATIONS (Cont'd.)

INDEX TO SYNTAX EQUATIONS (Cont'd.)