2/19

TECH MEMO a working paper

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

TM-2260/002/00

AUTHORS S. L. Kameny [illegible]
M. Levin [illegible]

TECHNICAL [illegible]
S. L. Kameny

RELEASE [illegible]
B. Barancik
for
D. L. Drukey

DATE 2/19/65 PAGE 1 OF 14 PAGES

LISP II PROJECT

Memo No. 1A

THE INTERNAL LANGUAGE

Abstract

This memo is Number 1A of series of working memos describing LISP II development. It replaces Memo 1 of the series and describes the internal S-expression language of LISP II to the extent to which it is now known. Certain features such as the Comit-type rule are deliberately omitted because they are covered elsewhere. Source language examples are given in most cases. The syntax of this memo is meant to be fairly complete, and is quite trivial. The semantics are incomplete and specify only what is directly related to the syntax.

GENERAL RULES

In internal language, any statement or expression is self-delineating in the form of an S-expression. Consequently, any expression in IL has the same format as a statement.* All primitive statements (those composed of a single S-expression and not containing a statement inside of them) have values, and may be used as expressions except for the statement (GO ⟨loc⟩) which has no value.

In source language, an expression is not self-delineating, and must be enclosed in brackets;

[⟨E⟩] or BEGIN⟨E⟩END

in order to be used as a statement.

1. Identifiers

Identifiers are composed of digits, letters, and @. The first character must be a letter. One is not free to incorporate additional non-ASCII characters into identifiers. However, strings may contain any legal characters of the hardware.

1.1 Reserved Words

It is important that the user does not have to worry about reserved identifiers any more than is absolutely necessary. In particular, the procedures local to the compiler must be invisible to him unless he wishes to use them by means of tailing symbols (see 1.2.).

Certain identifiers must be reserved because they are key syntactic words in the source language (e.g., BEGIN and ELSE). This list is quite short. The only other reserved words that the user need be concerned with are those that name important system functions that the user will actually need. Those that he doesn't need will be overridden by his own declarations.

The user is cautioned to stay away from all identifiers with @. These are used in the internal language for purposes that are not explicit in the source language (e.g., I@PLUS).

* However, any expression which is a pure function call is meaningless when used as a statement. A statement must have side effects, hence corresponds to a pseudo-function.

The user's instructions are:

1. Learn the list of reserved words. (About 40 words)
2. Don't use identifiers with @.

1.2. Tailing Symbols

In the source language. these appear as a sequence of identifiers with $ between them, and no spaces, for example, XI$COMPILER. The internal representation is (TAIL@ XI COMPILER) which is an S-expression. Every identifier except the last must be the name of a procedure or function, or the label of a block. The first identifier must have local significance. The rest of the list is then interpreted at the locality of the first. This process is repeated down the list, until the last item is found.

2. Expressions

2.1. Constants

If a constant is non-atomic, or if it is an identifier, then it must be quoted. If it is a string, number, or other type of atom, then it may or may not be quoted. The quoting is as in LISP 1.5.

2.2 Composition

Constants and variables are expressions. If fn is a procedure or a function, and if the e_i are expressions, then (fn e_1 ... e_n) where $n \geq 0$ is an expression.

2.3. Conditional Expressions

If the p_i and the e_i are expressions, then (COND (p_1 e_1) ... (p_n e_n) is an expression. Each p_i must properly be Boolean-valued. The behavior of non-Boolean values will not be guaranteed. The value of the conditional expression is the most general of the values of the e_i. The most general type is SYMBOL. The compiler will optimize by distributing transfer functions and cancelling where two composed transfer functions are the inverse of each other.

Example:

Source Language:

```
BOOLEAN B,D: INTEGER A,C: REAL B:
A ← IF B THEN C ELSE IF D THEN E ELSE F:
```

Internal Language:

(SET A (COND B C D E TRUE F))

2.4 The Assignment Statement

The assignment statement may be used as an expression. Its value is the value of its right half. In the source language, left arrow has lower precedence than the Boolean operators.

2.5 Subscripted Variables as Expressions

Source language: A[I,J]

Internal language: (FROM A I J)

(FROM A) less the value A

(FROM A e_i ... e_n) has the same meaning as the source language expression A[e_i ... e_n]

2.6 Bit Modified Expressions

Source language: BIT (A,I) or BIT (A,I,J)

Internal language: (BIT A I) or (BIT A I J)

The value of the expression (BIT A I) is an integer containing the I^{th} bit of the value of expression A right justified in an otherwise zero word.

The value of the expression (BIT A I J) is an integer containing J bits extracted from the value of expression A, starting at bit I, and right justified in a word. I and J are treated as integers. The expression is undefined if $I < 0$, or if $J < 1$ or if $I+J >$ word size.

2.7 BYTE Modified Expressions

Source language: BYTE (A,I) or BYTE (A,I,J)

Internal language: (BYTE A,I) or (BYTE A I J)

These have the same value as BIT(A, b*I, b*J) assuming the BYTE size to be b, and similar remarks apply.

BYTE has meaning as a locative in the left side of an assignment statement as well.

BYTE (A[I,J], K,L) etc. are all meaningful expressions if A can be subscripted.

2.8 Locatives

⟨locative⟩ ::= ⟨subscripted variable⟩ | ⟨variable⟩ | ⟨BIT modified variable⟩
⟨BYTE modified variable⟩

⟨locatives⟩ occur only at the left half of assignment statements. The complete set of all locatives is:

Source language	Internal language
n	n
n [x_1 ... x_n]	(n x_1 ... x_n)
BIT (n, i, j)	(BIT n i j)
BIT (n [x_1 ... x_n], i, j)	(BIT (n x_1 ... x_n) i j)
BYTE (n, i, j)	(BYTE n i j)
BYTE (n [x_1 ... x_n], i. j)	(BYTE (n x_1 ... x_n) i j)

in which only the most complicated form of the BIT and BYTE locatives have been shown. (The subscript i may be missing.)

Note that, by reason of its position in an assignment statement, a ⟨locative⟩ which is a subscripted variable does not require any flag, other than the fact that it is non-atomic, to tell that it is subcripted.

The BIT and BYTE forms are effectively ⟨locatives⟩ within ⟨locatives⟩.

Note that BIT and BYTE are reserved words and cannot be names of variables.

2.9 Designational Expression

⟨designational expression⟩ ::= ⟨label⟩ | ⟨subscripted label array⟩

A label array is also called a switch.

3. Statements

The types of statements are:

1. conditional statement
2. go to statement
3. for statement
4. assignment statement
5. procedure statement
6. compound statement
7. block

There is no reason why this list cannot be extended at some time. The Comit-type rule or equivalent will be included as a statement.

3.1 Labels

Source language: ⟨label⟩ : ⟨statement⟩

Internal language: ⟨label⟩ ⟨statement⟩

Any statement may be labeled, to any depth. It is meaningless and illegal to label an expression.

3.2 Compound Statements

Source language: BEGIN S_1: ... ;S_n END

or

[S_1; ... ;S_n]

Internal language: (PROG() S_1 ... S_n)

The value of a compound statement is the last statement executed, usually that of the executed RETURN statement. Labels in a compound statement are global to the context of the compound statement.

3.3 Blocks

Source language: BEGIN D_1; ... ;D_m;S_1; ... ;S_n END

or

[D_1; ... ;D_m;S_1 ... ;S_n|

Internal language: (PROG (D_1 ... D_m) S_1 ... S_n)

The context of a compound statement used as an expression is just that expression. The context of a compoun d statement used as a statement is the smallest block containing that statement.

It is possible to enter or leave a compound statement used as a statement, but is meaningless to enter or leave an expression.

e.g., in (FN (PROG () A S_1 S_2 (RETURN e_3)) e_4)

The expression S_2 may include a meaningful (GO A). However, the label A is completely unreachable outside of the compound statement.

The only difference between a block and a compound statement is the declaration list, which causes a pushdown brick to be assembled for this block.

Labels within a block are local to the block. Exit from a block is by means of a RETURN statement or by means of an EXIT statement, never by means of a GO statement.

3.4 Assignment Statements

Source language: ⟨locative⟩ ← ⟨expression⟩

Internal language: (SET ⟨locative⟩ ⟨expression⟩)

The stored quantity must be of a suitable type for the variable it is being stored into. Conversion functions will be invoked when necessary.

The left part of an assignment statement must be a ⟨locative⟩. (An actual parameter to be transmitted by LOC must also be a locative expression.) Simple variables and subscripted variables are locative expressions. Table operations (as yet undefined) may contain some locative expressions.

Assignment statements may be used as expressions in other assignment statements, for example:

Source language: Z ← Y ← X ← 2.0

Internal language: (SET Z (SET Y (SET X 2.0)))

Note that in this case there is only one expression to be evaluated, and this value is applied with proper conversions to each locative.

For a more complicated example, consider:

Source language: Z[I,J] ← Z[I,1] + (X ← Z[I,J])

Internal language: (SET (Z I J) (PLUS (FROM Z I 1) (SET X (FROM Z I J))))

Although there is only one possible way to parse the source language statement legally without parentheses in this case, the parentheses may be required for clarity when a statement is used as an expression.

Formal transmission of a parameter used as a left part will require some dynamics at run time.

3.5 Conditional Statements

There are two forms of conditionals in IL, namely the SELECT and COND forms. Each of these is similar to the corresponding forms in LISP 1.5, except that redundant parentheses have been removed. (Note that it is possible to tell the LISP 1.5 form of COND or SELECT from the LISP II form, so that both forms could be accepted as input language.)

COND form:

Source language: IF p_1 THEN S_1 ELSE IF ...

Internal language: (COND p_1 S_1 $\left\{p_i\ S_i\right\}^n$)

SELECT form:

Source language: SELECT (E ; E_1, S_1 $\left\{E_i,\ S_i\right\}^n$; S_x)

Internal language: (SELECT E. (E_1 S_1 $\left\{E_i\ S_i\right\}^n$) S_x)

In both the COND and SELECT forms, all S_i can be expressions, and the result is either a conditional expression or a conditional statement. If any of the S_i are not expressions then a conditional statement results.

In the SELECT statement, S_x may be missing from either format, but the final semicolon must appear in source language.

The unsatisfied conditional statement has no effect. The unsatisfied conditional expression causes an error.

3.6 Procedure Statement

Semantically, this is an expression which gets evaluated and its value, if any, is ignored. A procedure which does not have a value can only be used in this way.

3.7 For Statements

Source language:

$$\text{FOR } \langle\text{variable}\rangle \leftarrow \langle\text{for list element}\rangle \left\{, \langle\text{for list element}\rangle\right\}^{n} \text{ DO } \langle\text{statement}\rangle$$

$$\begin{aligned}\text{where } \langle\text{for list element}\rangle ::= {} & \langle\text{expression}\rangle \Big(\langle\text{empty}\rangle \mid \\ & \big(\text{MAPCAR} \mid \text{MAP} \mid \text{STEP} \mid \text{RESET}\big) \\ & \langle\text{expression}\rangle\Big) \Big(\langle\text{empty}\rangle \mid \big(\text{WHILE} \mid \text{UNTIL}\big) \\ & \langle\text{expression}\rangle\Big)\end{aligned}$$

Internal language:

$$(\text{FOR } \langle\text{locative}\rangle \left(\left\{ E_1 \left\{\begin{matrix}\langle\text{empty}\rangle \\ \text{MAPCAR} \\ \text{MAP} \\ \left.\begin{matrix}\text{STEP} \\ \text{RESET}\end{matrix}\right\} E_2\end{matrix}\right\}\right. \left\{\begin{matrix}\langle\text{empty}\rangle \\ \text{WHILE } \beta \\ \text{UNTIL } E_3\end{matrix}\right\} \right)^{n+1} \langle\text{statement}\rangle$$

This is a generalization of the ALGOL 60 For-statement, in that MAPCAR and MAP have the interpretation FOR X an element of list L or FOR X is a sublist of L.

Step is legal only if the ⟨variable⟩ and E_1 and E_2 and E_3 are all arithmetic. β is any predicate. The forms with RESET, MAPCAR, and MAP are valid for all variable types.

The value of a FOR statement is the exit value of the FOR variable ⟨locative⟩.

Note that the type of the FOR variable is a ⟨locative⟩ in the same types permitted in assignment statements, and must be declared in a block heading elsewhere.

Several variants of the FOR statement as well as several ideas on optimization are currently being considered. The key to recognition is the statement beginning (FOR ...).

3.8 Go To Statements

Source language:	GO A
	GO A[I,J]
Internal language:	(GO A)
	(GO (A I J))

In source language, GO TO is equivalent to GO. The form (GO (A I J) is used both to keep the syntax translation into internal language consistent with the locative translation of A [I J], and to make the interpretation of the GO statement easire.

The switch is a constant array of labels as in Algol 60.

4. Functions

A function is not an expression to be evaluated. The word function is used here as in LISP. The Algol term ⟨function designator⟩ is confusing, because it describes an expression, and should be ignored.

A function may be an identifier both in the source language and in the internal language, e.g., CAR.

4.1 Unnamed Functions Using LAMBDA

(LAMBDA (⟨parameter list⟩) ⟨expression⟩)

The elements of the parameter list may be atomic, in which case they are variables bound as parameters. If they are non-atomic, they also carry declarative information. For example:

(LAMBDA (X (R REAL)) ⟨expression⟩)

says that R is of the REAL whereas X is of type symbol

If there are no parameters, one must still write (LAMBDA () ...

4.2 Named Functions Using FUNCTION

The word FUNCTION combines the meaning of LAMBDA with that of LABEL in LISP 1.5.

```
(FUNCTION ⟨name⟩ (⟨parameter list⟩) ⟨expression⟩)
```

For example:

```
(FUNCTION FF (X) (COND (ATOM X) X TRUE (FF (CDR X))))
```

The name may specify a type for the value of the function, otherwise this is assumed to be SYMBOL.

```
(FUNCTION (SQUARE REAL) ((X REAL)) (TIMES X X ))
```

4.3 Array Allocation Function

Internal language: (ARRAY ⟨type⟩ $d_1 \ldots d_n$)

creates an empty (zeroed) array of n dimensions and bounds $d_1 \ldots$ and presets it to the array constant.

d_i can have as value either an integer or a dotted pair of integers

$d_i = n$ is equivalent to $d_i = (1.n)$

$d_i = (m.n)$ is legal if and only if m   n.

The value of (ARRAY ...) is the array itself.

5. Declarations

5.1 Parameter Declaration

Declarative information on parameters may include type or array type, and mode.

⟨type⟩ ::= REAL | INTEGER | BOOLEAN | OCTAL | SYMBOL | ...

⟨array type⟩ ::= ARRAY | ⟨type⟩ ARRAY

⟨mode⟩ ::= VALUE | LOC | FORMAL | FUNCTIONAL | GLOBAL | OWN

Examples:

```
(A REAL)
(B  BOOLEAN ARRAY))
(C INTEGER GLOBAL)
```

Parameters may not be OWN.

5.2 Functions with an Indefinite Number of Arguments

It is possible to define a function with an indefinite number of arguments. There must be only one set of such arguments, all of the same type, and transmitted by VALUE. There may be other arguments that come first. The formal parameter list for such a function has as its last parameter something like (AN). This causes transmission of an array A containing the indefinite arguments and an integer N specifying the size of A. The calling sequence ignores this fact.

Example:

(SUMSQUARE A B C D) is a call to a function that computes $\sum a^2$.

Its definition is:

```
(FUNCTION (SUMSQUARE INTEGER) ((A INDEF N))

(PROG (U V)

(FOR V (1 STEP 1 UNTIL N) (SET V (PLUS (SQUARE (FROM A U)))))

(RETURN V)))
```

In source language, this appears as

```
INTEGER FUNCTION SUMSQUARE (A[N]); INTEGER U, V:

BEGIN FOR U ← 1 STEP UNTIL N DO V ← V + SQUARE (A[U]);

RETURN (V) END
```

5.3 Program Variable Declaration

Program variables are declared in a list following PROG which has the same format as the list following LAMBDA. Program variables can not be LOC, FORMAL, or FUNCTIONAL.

Program variables (but not parameters) may be initialized to some value upon entry. This is indicated by an additional element on the declaration list.

Examples:

```
(A REAL 3.4)

(B INTEGER (TIMES U V))
```

When there is no explicit initialization, program variables are set to NIL, FALSE, 0, 0.0 etc. according to type.

5.4 Macro Definition

(MACRO ⟨name⟩ (⟨argument name⟩) ⟨expression⟩)

MACRO creates a named macro, which is a function of one argument, namely the quoted S-expression whose CAR is equal to the name of the macro. The value of the macro is the macro expression applied to the macro call.* (This agrees substantially with Tim Hart's macro definition.)

Example:

```
    (FUNCTION EXPAND (L OP)

       (COND (NULL (CDDR L)) (CADR L) TRUE

           (LIST OP (CADR L) (CONS CAR L) (CDDR L)))))

    (MACRO PLUS (J) (EXPAND J (QUOTE PLUS2)))
```

(in which PLUS2 is a function of two arguments)

would define PLUS as a macro which adds an indefinite number of arguments. PLUS could also be defined as a FUNCTION of indefinite number of arguments in LISP II.

6. ERSET

ERSET is a way of handling abnormal exits that cross declaration bounds. The ERSET statement is

Source language: ERSET ⟨locative⟩ THEN ⟨statement⟩

Internal language: (ERSET ⟨locative⟩ ⟨statement⟩)

It is turned off by

Source language: ERREST

Internal language: (ERREST)

It may be used at successive levels.

Nothing happens until EXIT (⟨ Σ ⟩) is called. When this is called, Σ is evaluated. Then the push down stack is unwound to the level of the innermost ERSET. The locative in ERSET is set to the value of EXIT, and the statement is then executed.

If EXIT is used without ERSET, it will unwind to the top interpretive level which is LISTEN ().