Mr. C. Weisman

LISP II Project
Information International, Inc.
Memo 1

1-177

THE INTERNAL LANGUAGE

By

Michael Levin

ABSTRACT

This memo 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 have yet to be determined. 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. This memo does not specify data types.

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 hardward.

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).

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, X1$COMPILER. The internal representation is (TAIL@ X1 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 E;
A←IF B THEN C ELSE IF D THEN E ELSE F;
```

Internal language:

```
(SETQ A (COND (B C) ( D E) (TRUE F)))
```

After insertion of transfer functions:

```
(SETQ A (SICON@ (COND (B (ISCON@ C)) (D (RSCON@ E)) (TRUE F) )))
```

After distribution and cancellation of transfer functions:

```
(SETQ A (COND (B C) (D (RICON@ E)) (TRUE (SICON@ F)) ))
```

2.4. The Assignment Statement

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

2.5. Subscripted variables

Source language: A[I,J]

Internal language: (ARRAY A I J)

3. Statements

The types of statement 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 eventually be included as a statement.

3.1. Labels

Source language: ⟨label⟩ : ⟨statement⟩

Internal language: (LABEL ⟨label⟩ ⟨statement⟩)

3.2. Compound Statements

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

or

[S_1; ... ;S_n]

Internal language: (COMP@ S_1 ... S_n)

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: (BLOCK@ (D_1 ... D_m) S_1 ... S_n)

If there are no declarations, then there must still be a null list of declarations.

3.4. Assignment Statements

Source language: ⟨left part⟩ ← ⟨expression⟩

Internal language: (SETQ ⟨left part⟩ ⟨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. (See example 2.3.)

The left part of an assignment statement must be a locative expression. (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.

Formal transmission of a parameter used as a left part will require some dynamics at run time. The notation FUNC@ indicating a function to be applied at run time may be inserted during the next stage of compilation.

3.5. Conditional Statements

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

Internal language: (COND (p_1 S_1) ...)

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

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 TO ⟨designational expr.⟩

or

GO ⟨designational expression⟩

Internal language: (GO ⟨designational expression⟩)

The word TO is ignored.

The designational expression is like any other expression. The label is a type. However, there are no conversion functions between labels and other types (unless the integer label is resurrected). The only thing to do is to go to it. SWITCH is a synonym for LABEL ARRAY. They may be multi-dimensional. It is permissible to assign a label to a switch, whereas the Algol 60 switch is a constant array. Labels and switches may be used as parameters. Non-local labels must be referred to by tailing.

It is not permitted to go into or out of a procedure. It is permitted to go into or out of a compound statement.

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.

There is a distinction between a function and a procedure in LISP II. The function has an expression as its body. The expression is evaluated and this is the value of the function. The procedure has a statement as its body. The statement is executed. If the procedure has a value, then this is obtained either by RETURN, as in the LISP 1.5 PROG, or by the most recent value assigned to the procedure identifier from within itself as in the Algol 60 procedure.

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

A function may be labeled, in which case it is still a function.

Source language: ⟨label⟩ : ⟨function⟩

Internal language: (LABEL ⟨label⟩ ⟨function⟩)

A function may be designated by a LAMBDA expression. The ASCII character for λ is @.

Source language: @ (X_1, ... ,X_m);D_1; ... ;D_n;E

Internal language: (LAMBDA (X_1 ... X_m) (D_1 ... D_n) E)

where the X_i are formal parameters, the D_i are declarations, and E is an expression. In the internal language, the declaration list must always be present even if it is null.

The internal language conventions thus are exactly those of LISP 1.5 with the insertion of the declaration list. For LISP 1.5 functions using slow arithmetic, this will be null.

4.1. Procedures

The syntax is identical to that for functions except that the symbol @ in source language, and the identifier LAMBDA in internal language are replaced by the identifier PROCEDURE. The procedure body is a statement rather than an expression.

4.2. Values of Functions and Procedures

The type of value returned by a function or procedure must be known at compile time if efficient arithmetic programs are to be generated. The type of value may be declared or assumed to be SYMBOL.

Source language:
```
TYPE REAL;
TYPE INTEGER ARRAY;
```

Internal language:
```
(TYPE REAL)
(TYPE INTEGER ARRAY)
```

5. Declarations

Lists of declarations occur in the headings of blocks, procedures, and functions. In the source language, they are separated by semicolons; in the internal language, they are elements of a list.

5.1. Answer Type Declarations

The answer type declaration in the procedure or function defines the type of the answer returned. (See 4.2.)

5.2. Mode Declarations

These specify the mode by which a parameter is transmitted.

Source language: VALUE X, Y; LOC Z; FORMAL W;

Internal language: (VALUE X Y) (LOC Z) (FORMAL W)

5.3. Type Declarations

These specify the data type of parameters and variables.

Source language: REAL X, Y; SYMBOL Z;

Internal language: (REAL X Y) (SYMBOL Z)

5.4. Type Declarations with Initialization

A type declaration may also cause initialization of variables (but not parameters).

Source language: REAL X, Y←2.0, Z; SYMBOL W←'(A . B);

Internal language: (REAL X (Y 2.0) Z) (SYMBOL (W
(QUOTE (A . B))))

5.5. Storage Specification

The storage modes are LOCAL (on the push down stack and accessible only locally), GLOBAL (in a special location and accessible anywhere), and OWN (statically declared, unrecursive, and accessible locally or by tailing).

Source language: GLOBAL X; OWN Y;

Internal language: (GLOBAL X) (OWN Y)

5.6. Array Declarations

Since arrays are data in LISP II, it is not always necessary to declare a size. One may declare that a given variable is an array variable, and later place an array in it. In this case, it is not meaningful to subscript the variable until an array has been created. These alternatives execute at various speeds with the static array being the fastest. OWN arrays must be statically declared with bounds. Arrays must have a data type specifying what type of data are stored in them. Arrays may be initialized.

Source language: REAL ARRAY W[2,2]←[[3.0,4.7],
[6.0,4.2]]

Internal language: (REAL ARRAY ((W 2 2) [[3.0,4.7],
[6.0,4.2]]))

5.7. Mixed Declarations

Any of the previous declarations may be combined when meaningful.

Source language:
```
GLOBAL LOC INTEGER X;
OWN REAL Y←3.0;
GLOBAL INTEGER X←U*V;
```

Internal language:
```
(GLOBAL LOC INTEGER X)
(OWN REAL (Y 3.0))
(GLOBAL INTEGER (X (TIMES U V)))
```

5.8. Default Assumptions

1. All data types are SYMBOL.
2. All function and procedure values are of type SYMBOL.
3. All parameters are transmitted by VALUE.
4. All parameters and variables are LOCAL.
5. All arrays are of type SYMBOL.
6. All unspecified initializations are NIL, 0, 0.0, FALSE, etc.

5.9. Function and Procedure Declarations

It is possible to declare a variable whose value is a function or procedure:

Source language:
```
REAL PROCEDURE X;
```

Internal language:
```
(REAL PROCEDURE X)
```

Like any other variable definition, this may be initialized to some function:

Source language:
```
FUNCTION SECOND←@(X); CADR (X);
```

Internal language:
```
(FUNCTION SECOND (LAMBDA (X) () (CADR X)))
```

In the source language only, there is an alternative syntactic convention that is compatible with Algol 60. This consists of merely eliminating the left arrow and the symbol immediately following it. This causes no loss of information.

Source language:
```
FUNCTION SECOND (X); CADR (X);
```

The procedure declaration is the same as the function declaration except that the LAMBDA or @ is replaced by PROCEDURE.

Note that in such a declaration, the need for an answer type declaration within the function disappears. A REAL PROCEDURE must define a procedure with answer type real.

My apologies for a hasty and undebugged memo. I wanted everyone to think about these things before the next meeting. I'm sure there will be lots to object to then.

1/26/65