LISP A: A lisp-like system for incremental computing

by ERIK J. SANDEWALL
Uppsala University
Uppsala, Sweden

BACKGROUND: THE LISP LANGUAGE

The work reported here is based on the LISP 1.5 programming language. Let us therefore begin with a short review of LISP 1.5. Those who have used LISP would be able to proceed directly to the section OUTLINE OF THE SYSTEM.

All *data* in LISP are in the form of S-*expressions*, i.e., correctly parenthesized expressions, whose ultimate components (*atoms*) are similar to FORTRAN variables or constants. Thus

((A 3.14 BETA) ((GAMMA) (T345)))

is an example of an S-expression. Its sub-expressions, e.g.,

(A 3.14 BETA)

(GAMMA)

etc., down to and including the atoms A, 3.14, etc. are also S-expressions.

Inside the computer, each S-expression is represented as a list structure, i.e., as a system of address references between cells in core. This need not worry us here.

For each atom, there is a *property list*, on which characteristic facts about the atom may be stored. One atom may have several properties, and they are therefore distinguished through the use of *attributes*. A property-list is essentially a collection of attribute-property pairs. Any atom may be used as an attribute, and any S-expression as property.

So much for data. The *program* in a LISP job is also written as an S-expression. This makes it possible for a program to modify itself. The program is usually interpreted at run time, rather than compiled.

The format for writing expressions in the program is exemplified through

(PLUS A 3.14 (MINUS C) (TIMES F (PLUS G H)))

which would mean the same as the FORTRAN expression

A + 3.14 + (−C) + F * (G + H)

Thus functions in LISP are always prefixed to their arguments, and always stand immediately after a left parenthesis. Usually, functions for manipulation of S-expressions dominate in a LISP program, and arithmetic functions like the ones in the example are only sporadically used.

There are two ways to write the program, either as a FORTRAN-like, step by step program, or as one single expression. In the former case, we may use atoms like statement numbers, and write GOTO statements in the format of (GO POSITION). In the latter case (which is considered more "pure"), conditional expressions and recursivity are used, and virtually eliminate the need for assignment statements, goto statements, etc. The statement structure collapses, and the "program" degenerates into one single expression, that is to be evaluated.

The present work is based on the latter type of "program".

As a subroutine-like feature, it is possible to define in LISP new functions in terms of elementary or previously defined ones. Suppose F1, F2, and F3 are defined as functions, and we want to define G as a function in such a way that the value of

(G X Y)

is identical to the value of

(F1 (F2 X) (F3 Y))

for any X and Y. Through the use of a certain elementary function, we put the S-expression

(LAMBDA (X Y) (F1 (F2 X)(F3 Y)))

on the property-list of the atom G and (usually) under the attribute EXPR. The atom LAMBDA plays a special role in the system, and must always be there.

It is important to distinguish between the *S-expression*

(F1 (F2 X)(F3 Y))

on one hand, and *the value* of this expression, when it is used in a (or as a) program, on the other hand. In the former case, we are referring to a piece of data, which is passive, and which is only potentially a program. In the latter case, we are referring to the output of a program. In this paper, we shall always let e.g.,

(F 1 (F2 X)(F3 Y))

refer to the S-expression itself, not to its value. If we want to refer to the value, we shall write either

"the value of the S-expression (F1 (F2 X)(F3 Y))"

or, conforming with mathematical notation,

f1(f2(x), f3(y)).

Note that the function symbols are now outside the parentheses that enclose the argument(s).

For further information about LISP 1.5, the reader is referred to John McCarthy's original article,[1] to the LISP 1.5 manual,[2] or to a recent textbook.[3] It may also be worthwhile to study some programs that have been written in LISP, as reported in e.g.[4,5,6] What has been said here should however be sufficient to get some grasp of the present paper.

Outline of the system

LISP A is an extended and modified version of LISP 1.5. Its main characteristics are:

1. Atoms for sets of objects

If an atom has certain properties on its property-list, the system recognizes it as a symbol for a set of objects. Such an atom is called an *ambject*. For example, we might have an ambject BOY for "the set of all boys".

As ambjects stand for sets, their property-lists may e.g., contain references to other ambjects that stand for subsets or supersets of the given set.

2. New: symbolic functions

Besides the types of functions used in LISP 1.5 (i.e., machine-coded SUBR's and FSUBR's, and LAMBDA-expression EXPR's and FEXPR's), LISP A recognizes *symbolic functions*, which are evaluated in a special way. Let FATHER be a symbolic function. When LISP A evaluates the form

(FATHER JOHN)

it assumes the atom JOHN to have an ambject as value,[*] and returns as value a new ambject which has the "meaning" list (FATHER JOHN) on its property-list under the attribute MEANING.

Symbolic function are used for building up data structures. They can be combined freely with other types of functions. Thus it could be sensible to evaluate the expression

(DESCRIBE (FATHER JOHN)).

The symbolic function FATHER creates an ambject for "the father of John" (unless such an ambject is already there, in which case it is retrieved, rather than re-created). The ordinary function DESCRIBE is defined through ϵ LAMBDA-expression, and it digs into the data structure to create a description of its argument.

Likewise, it might be sensible to evaluate the S-expression

(SETTRUE (ISFATHER DICK JOHN)).

Here, ISFATHER is a symbolic predicate, which is a kind of symbolic function. An ambject generated by a symbolic predicate can have an indication about truth or falsehood on its property-list. The ordinary function SETTRUE sets an indication that its argument is true, and it may also be defined to perform some inference from this fact.

3. New:RHO-expressions

There is still another type of functions, namely RHO-expressions. They have in principle the same form as LAMBDA-expressions, but they work quite differently. If

$$fn = \rho((x,y)\ g(x,y))$$

and (FN A B) is evaluated with A and B ambjects, then the system will essentially apply the function G to each combination of members of A and members of B. In other words, (FN A B) performs: "for each X in A and each Y in B, evaluate (G X Y)". The facts "X is in A" etc. must of course be retrieved from the property-lists of the ambjects A and B.

* It is convenient to make the convention that all ambjects have themselves as values. Thus when the system evaluates
    (FATHER JOHN)
it first evaluates the atom JOHN, which is an ambject and has itself as value. This means that
    father(john)
and
    father(JOHN)
are identical.

4. RHO-expressions are triggered automatically.

The system takes two actions when evaluating an expression like (FN A B) above:

41. It evaluates (G X Y) for each X that is presently known to be in A, and each Y that is presently known to be in B.
4.2 It stores away the expression (FN A B) in the data base. Subsequently, each time a new Xi is added to the cases of A, the expression (G X_i Y) is evaluated for each Y in B. The symmetric action is taken when a new Y_j is added to the cases of the B.

For example, we might evaluate

ρ((x) print(describe(x))) (cousin(Peter))

where DESCRIBE has been defined before;

COUSIN is a symbolic function. cousin (Peter) is an ambject that stands for the set of all Peter's cousins;

and the RHO-operator will cause a description to be printed for each one of Peter's cousins that the system has recognized, and each cousin that it later recognizes.

Another example is

ρ((x) settrue(admire(x,father(x))))
(boy $\cap$ discussed-obj)

where FATHER and BOY have been defined before;

ADMIRE is a symbolic predicate;

SETTRUE has been defined before;

DISCUSSED-OBJ is the set of all objects that are currently interesting to the system. This is a kind of heuristic information.

For each boy who is also a discussed object, this operator will assert that the boy admires his father. The situation might be that every boy admires his father, but that we only want to execute the inference for those boys that attract our interest in particular.

In these examples, the RHO-expressions were used for their side-effects. After we have given an interpretation of rho-expressions in terms of the sets they take as arguments, we shall be able to evaluate them for their value as well. This is discussed in section SEMANTICS OF RHO-EXPRESSIONS.

5. Expressions can be treated as objects

The symbolic function SYMQUOTE makes it possible to form sets of expressions, and to manipulate those sets. SYMQUOTE creates an ambject, that stands for another ambject. (Does this sound confusing?) Remember that the value of e.g. the S-expression

(FATHER JOHN)

is an ambject that stands for John's father. The value of the expression

(SYMQUOTE (FATHER JOHN))

is an ambject, that stands for the ambject, that stands for John's father. Iterated quoting is permitted (but probably rarely useful).

The function SYMQUOTE should be distinguished from the function QUOTE in ordinary LISP. (QUOTE is also defined in LISP A). To take an example, the value of the expression

(QUOTE(FATHER JOHN))

is the S-expression

(FATHER JOHN).

This S-expression then has not been evaluated: The purpose of QUOTE is to prevent its argument from being evaluated, and to return it as it is. On the other hand, if we evaluate

(SYMQUOTE(FATHER JOHN))

we do first evaluate

(FATHER JOHN)

and obtain an ambject as value. After that, SYMQUOTE takes the argument and creates a new ambject just like any other symbolic function.

There is one important difference between SYMQUOTE and other symbolic functions, however. This has a philosophical parallel. If we know that Dick and John's father are identical persons, we can conclude that the wife of Dick and the wife of John's father are also identical. However, we can not claim that the *linguistic expression* "Dick" and "John's father" are identical. One consists of four letters, the other of eleven.

The function SYMQUOTE performs exactly that kind of quoting. Thus for any symbolic function fn *except symquote*, we would have

$$a = b \supset fn(a) = fn(b)$$

In conclusion, the value of the expression

(SYMQUOTE DICK)

is an ambject that stands for the ambject Dick; the value of the expression

(SYMQUOTE (FATHER JOHN))

is another ambject and stands for the ambject father (John).

SYMQUOTE can be used to speak about expressions that are already in the data base. For example, it makes sense to form the set of all ambjects obtained from the evaluation of

```
(FATHER JOHN)
(FATHER LUCIA)
(FATHER PETER)
```

etc. If this set is called FATHEREXPRESSION, it would be correct to evaluate e.g.

```
(SETMEMBER (SYMQUOTE (FATHER JOHN))
    FATHEREXPRESSION ).
```

Here, SETMEMBER is an ordinary function that declares its first argument to be a member of its second argument. It can be compared to SETTRUE which was used above.

6. Thanks to SYMQUOTE, LISP A "knows what it is doing", each time it uses a symbolic function.

If FATHER is a symbolic function like before, and the system evaluates the form

```
(FATHER JOHN),
```

it will create (or retrieve) two ambjects:

6.1 An ambject fj with the meaning "John's father",

6.2 An ambject qfj with the meaning "the ambject fj".

The ambject qfj is identical to what would have been obtained as the value of

```
(SYMQUOTE (FATHER JOHN)).
```

The LISP A system returns fj as the value of

```
(FATHER JOHN),
```

but before that, it has automatically performed the operation

```
(SETMEMBER (SYMQUOTE (FATHER JOHN))
    FATHEREXPRESSION).
```

In this way, each RHO-expression that has previously been evaluated with FATHEREXPRESSION as one of its arguments, will now be given a chance to operate on qfj. In other words, the system is able to take care of operators that say "whenever you see an expression where FATHER occurs as the leading function symbol, do the following: –". The action done may be some logical inferences, or some external action like a printout.

7. LISP A is an incremental computer

In ordinary use, the behavior of the LISP A system is governed by the RHO operators. The evaluation of one operator may trigger another, which in its turn triggers several other operators. (The question whether the resulting trees shall be handled on a depth-first or a parallel basis, is discussed in section IMPLEMENTATION). All operators are given to the LISP A system independently, and they communicate through the changes they make in the data base (= the system of ambjects). There is no single, deterministic program. For these reasons, we characterize the LISP A system as an *incremental computer.* (This term was introduced in an article by Lombardi and Raphael.[7] For a discussion of their system, see section OTHER INCREMENTAL COMPUTERS, (towards the end of the paper).

8. LISP 1.5 is (almost) a subset of LISP A

Some small and unimportant changes have been made to LISP 1.5, but in general, everything that can be done in LISP 1.5 can be done on almost identically the same form in LISP A. The differences are described in section DIFFERENCES FROM LISP 1.5.

Examples of ambjects and symbolic functions

Before we proceed, let us specify the interpretation of ambjects and symbolic functions more in detail.

Rule 1. All ambjects in the LISP A system stand for sets that consist immediately of objects. There is no obvious way to represent the objects themselves as ambjects, nor to represent sets of sets.

Example. The ambject ESKIMOO would stand for the set of all individual eskimoos.

Example. The ambject JOHN would stand for the set whose only member the person John is.

Rule 2. All symbolic functions take sets as arguments and yield new sets as values.

Example. The ambject

$$\text{father(JOHN)}$$

stands for the set whose only member John's father is.

Example. The ambject

$$\text{father(JOHN} \cup \text{NILS)}$$

stands for the set that has John's father and Nils' father as members. If John and Nils happen to be brothers, the set has of course only one member.

Example. Let the symbolic function DRIVERS be defined in such way that, if CAR4 stands for the set of one particular car, then drivers(CAR4)

is the set of all persons that ever drove this car. If X-CO-CAR is the set of all cars owned by X Co., then

drivers(X-CO-CAR)

is the set of all persons that ever drove some of their cars.

Example. Let the symbolic function BELONGINGS be defined such that e.g.

belongings (JOHN)

stands for the set of all objects that belong to John. If CAR stands for the set of all cars in the world, and X-CO for the set whose only member X Co. is, then we have

$$\text{X-CO-CAR} = \text{CAR} \cap \text{belongings(X-CO)}$$

and therefore of course

$$\text{drivers (X-CO-CAR)} = \text{drivers(CAR} \cap \text{belongings(X-CO))}$$

Remark. The functions FATHER, DRIVERS, and BELONGINGS all satisfy

$$\text{fn}(a \cup b) = \text{fn}(a) \cup \text{fn}(b)\ ;$$

this rule is characteristic of symbolic functions.

One predicate(*) is important. It takes one argument and says that its argument is a set of *exactly* one member. Clearly, we have

$$*a \supset *\text{father}(a)$$

but not

$$*a \supset *\text{drivers}(a)$$

nor either

$$*a \supset *\text{belongings}(a).$$

Remark: The fact that the LISP A system only "thinks about" sets of objects, never objects themselves, means that we have to reinterpret some statements made in section OUTLINE OF THE SYSTEM. Thus the function SETMEMBER must be defined in such a way that evaluation of

(SETMEMBER A B)

asserts that $*a \wedge a \subseteq b$.

Semantics of RHO-expressions

Like before, let

$$f = \rho((x,y)\ g(x,y)).$$

We shall interpret

$$f(a,b) \text{ as } \cup_{r\epsilon a} \cup_{s\epsilon b}\ g(\{r\}, \{s\})$$

In plain words: Let a and b be sets. Select an arbitrary r that is a member of a, and an arbitrary s that is a member of b. Form x = the set whose only member r is; y = the set whose only member s it. Construct the object g(x,y). This is a new set. Form the union of all such sets. The result is f(a,b).

From this, it immediately follows:

$$*a \wedge *b \supset \rho((x,y)g(x,y))(a,b) = g(a,b) \qquad \text{(R1)}$$

$$f(a_1 \cup a_2, b) = f(a_1,b) \cup f(a_2,b) \qquad \text{(R2)}$$

if f is a RHO-expression, and similarly for the second argument.

Consequently,

$$a_1 \subseteq a_2 \supset f(a_1,b) \subseteq f(a_2,b) \qquad \text{(R 3)}$$

if f a RHO-expression

This interpretation immediately generalizes to other rho-expressions with an arbitrary number of arguments. With this background, it is clear that the following ways of handling set-inclusion, RHO-expressions, etc., are sound:

1. Each ambject A has properties on its property-list under the following attributes:

SUBSET	The corresponding property is a list of all subsets of A(*). This includes A itself.
SUPERSET	The corresponding property is a list of all sets that have A as a subset. This includes A itself.
STAR	A flag which, if present, indicates that A has exactly one member.
OPERS	The corresponding property is essentially (see below) a list of all expressions on the form (FN ... A ...) where the leading function, FN, is a RHO-expression, and A occurs as one argument.

2. To evaluate a form whose leading function is a RHO-expression, we do as follows:
   2.1 Evaluate the arguments. They should all yield ambjects A1, A2, ... Ak.
   2.2 Create an ambject that is later to be returned as the value of the whole form. Give it a suitable MEANING property. (2.1-2.2 is the same treatment as is given forms with symbolic functions).
   2.3 Add the ambject created in (2.1) to the OPERS property of each argument A1, A2, ... Ak. (To be precise, the OPERS property of an ambject A is therefore a list of ambjects, each of which has a MEANING property which is a form with a RHO-expression as leading function and A as one of the arguments.)
   2.4 Consider each argument Ai. Its SUBSET property is a list of ambjects Ai1, Ai2, ... Aiki. Each such Aij stands for a subset of Ai. By checking for the occurrence of a STAR flag, select those Aij which have exactly one member, i.e., which satisfy STAR flag, select those Aij which have exactly one member, i.e., which satisfy *Aij.

   By forming various combinations of such Aij, we now notice that from law A1 above, i. e.

$$*a \wedge *b \supset \rho((x,y)g(x,y))\,(a,b) = g(a,b)$$

   follows

$$*a \wedge *b \supset \rho((x,y)g(x,y))\,(a,b) = \lambda((x,y)g(x,y))\,(a,b)$$

   We generalize this from two arguments a,b, to k arguments A1, A2 ... Ak, and decide on the following method:
   2.5 Evaluate all expressions

   $(FL\ A1m_1\ A2m_2 \ldots Akm_k)$

   where FL is the original RHO-expression, except that the RHO has been changed into a LAMBDA, and each Aim_i is a one-member subset of the argument Ai.
   2.6 In step 2.5, we obtain one ambject as value for each possible combination of one-member subset. By virtue of rule R2 above, each such ambject is a subset of the ambject created in (2.2). Mutual references are therefore put on the SUPERSET viz. SUBSET properties.
   3.1 Checks whether the fact $a \subseteq b$ is already known (i.e., whether the ambject be is already on the SUPERSET property of a). If so, it returns; otherwise, it continues.
   3.2 Adds each member b′ of the SUPERSET property of b (including b itself) to the SUPERSET property of a. If we know *a (i.e. if a has the flag STAR), we also apply the operators of b′ to a; see step 3.4.
   3.3 Adds each member a′ of the SUBSET property of a to the SUBSET property of b. If we know *a′, we also apply the operators of b to a′, see step 3.4. Then return.
   3.4 To apply the operators of b″ to a″, first retrieve the OPERS property of b″. It is a list of ambjects whose MEANING properties are forms

   (F1 ... ... B″ ...)

   F1 is a RHO-expression. It would be possible and legal to re-evaluate these RHO-expressions as described in step 2.4-2.6.* In step 2.5, where we form all combinations of one-member subsets, we would then obtain combinations where a″, the one-member subset of b″, is included, which is what we desire. However, we would *also* obtain combinations where other subsets than a″ are used, and these combinations have already been considered. To avoid this inefficiency, we first put the SUBSET property of b″ on the push-down-list, and replace it with another property that *only contains a″*. After that, all forms on the OPERS property of b″ are evaluated, and finally the old SUBSET property is restored from the push-down-list.
4. There is a function SETSTAR, which is similar to SETSUBSET. It puts the flag STAR on its sole argument, and triggers the necessary operators. The details are analogous to those for SETSUBSET.

The above algorithm can be considered as an encodement of the rules R1 -R3 for RHO-expressions and the * predicate, as given at the beginning of this section. As each occurrence of a symbolic function is reported via SYMQUOTE and may trigger operators, axioms for symbolic functions and predicates may be encoded as RHO operators.

Logic with four truth-values

We agree that a symbolic predicate should be a special kind of symbolic function. It is desirable that

* More precisely, is a list of all ambjects, that stand for subsets of the set that A stands for.

(*)Step 26 must be slightly modified; the ambject created in step 25 is now to be a subset of the ambject that carries the RHO-expression i.e. the ambject on b″:s OPERS property.

the difference between predicates and other functions should be kept as small as possible. In particular, to make it possible to use predicates inside RHO-expressions, we should let the value of a symbolic predicate be a *set of truth-values*.

Let t and f be the original truth-values, defined on relations between objects. Introduce the sets

$$\begin{aligned} T &= \{t\} \\ F &= \{f\} \\ S &= \{t,f\} \text{ (stands for “sometimes”)} \\ &= \{\ \} \end{aligned}$$

It easily follows

$$^{*}p \wedge p \subseteq T \supset p = T \qquad \text{(R4)}$$

This rule is important and will be used below.

We now extend the ordinary logical connectives to this four-valued logic. The connectives shall satisfy the general axiom fn(a ∪ b) = fn(a) ∪ fn(b), so we have e.g.

$$\begin{aligned} T \vee T &= \{t\} \vee \{t\} = \{t\} = T. \\ T \vee S &= \{t\} \vee \{t,f\} = \\ &\quad \{t\} \vee (\{t\} \cup \{f\}) = \\ &\quad (\{t\} \vee \{t\}) \cup (\{t\} \vee \{f\}) = \ldots \\ &\quad T \cup T = T \end{aligned}$$

Let us not here delve further into this logic. One of our early examples of the use of RHO-expression was

ρ((x) settrue(admire(x,father(x))))

(boy ∩ discussed-obj).

If we assume the very reasonable axioms

*a ⊃ *father(a)

and

*b ∧ *c ⊃ *admire(b,c),

we can re-write the operator on the equivalent form

settrue(ρ((x) admire(x,father(x)))

(boy ∩ discussed-obj)).

If we evaluate the operator in this latter version, the following will happen (although not necessarily in this order):

1. The ambject boy ∩ discussed-obj is evaluated and assigned some properties by operators that are triggered by the use of the function ∩.
2. The RHO-expression is applied to its argument. A new ambject, af, is introduced.
3. The function SETTRUE is applied to af, which is then set equal to the set T through a property.
4. Let r be an ambject which has obtained the flag STAR, and which has the ambjects BOY and DISCUSSED-OBJ on its SUPERSET property. When the last of these three conditions becomes satisfied, the above RHO-expression will be triggered. The system evaluates father(r) and admire(r,father(r)). These will of course be new ambjects. If the above axioms are properly encoded, the system will put the flag STAR on them. Moreover, by the specification of how to handle RHO-expressions, the system will evaluate

   setsubset(admire(r,father(r)), af),

   where af is the ambject introduced in step (2). By rule (R4) above, the ambject admire(r,father (r)) will be set EQUAL to af and therefore to T.

This was one example of how RHO-expressions can sometimes be evaluated for their value, rather than for their side-effect.

Differences from LISP 1.5

Through the introduction of new types of functions, the traditional means of handling functions in LISP become inconvenient. We found an alternative system of conventions, which may have some interest in itself.

The association-list in ordinary LISP assigns values to atoms. The value may be a FUNARG - expression (in which case the atom can be used as a function symbol) or an arbitrary S-expression (in which case the atom can only be used as the argument of some function).

On the association-list, it does not matter whether an atom stands for a function or something else, but in other parts of the LISP system it does. On property-lists, functions are defined as EXPR-properties or FEXPR-properties; other values as APVAL-properties. If the atom G has the EXPR-property gg, then the two expressions

(FUNCTION G) and
(FUNCTION gg)

are equivalent, but if the atom A has the APVAL-property (aa), the two expressions

(QUOTE A) and
(QUOTE aa)

are not at all equivalent.

In LISP A, the distinction between functional and other values is abolished. This leads to the following consequences:

1. The pseudo-function FUNCTION is superflous. We use QUOTE instead.
2. When LAMBDA-expressions are used directly in forms, they must be quoted. To avoid con-

fusion, we introduce the symbol ETA to be used instead of LAMBDA. Thus the following would be a correct form:

```
((QUOTE (ETA (X Y) (........)))

(...... .....)

(............))
```

3. When function definitions (e.g. LAMBDA-expressions) are named by atoms, they are put as APVAL-properties (viz. as VALUE-properties in PDP-6-LISP-type implementations).
4. Different kinds of functions, which were previously distinguished through their attributes (EXPR, FEXPR, SUBR, etc.) must now be distinguished some other way. We use the following transformations:
   4.1 A previous EXPR on the form
   (LAMBDA a b)
   is re-written on the form
   (ETA a b)
   4.2 A previous FEXPR on the form
   (LAMBDA (a1 a2) b)
   is re-written on the form
   (PHI a1 b).
   4.3 A previous SUBR is re-written on the form
   (ECODE . a)
   where a is the starting address for the machine code routine.
   4.4 A previous FSUBR is re-written on the form
   (FCODE . a)
   with a as for ECODE.
5. The general evaluation function for LISP A, evala, evaluates the leading function of a form just like any of the arguments. It therefore becomes possible to evaluate the expression for a function immediately before it is used. For example, we can write

   ((DERIVATIVE SINE) X)

   where (DERIVATIVE SINE) evaluates into the value of COSINE(*) which is then applied to the value of X.

The above changes have the obvious advantages of preparing the ground for the two types of functions in LISP A: symbolic functions and RHO-expressions.

(*) The value of the atom SINE is most likely an expression (ECODE . a). The function DERIVATIVE takes this as argument and uses it in an expression of the type

```
derivative (f) =
    if ......   else
    if f=sine then cosine else   ......
```

The value of (DERIVATIVE SINE) is therefore an expression (ECODE . b), which also happens to the value of the atom COSINE.

To increase compatibility with LISP 1.5, it is possible to define functions LAMBDA and LABEL as PHI-expressions. If we let the value of the atom LAMBDA be

(PHI R (CONS (QUOTE ETA) R))

we can use LAMBDA-expressions as leading functions in forms, just like in LISP 1.5 (LAMBDA-expressions that were EXPR's or FEXPR's must of course still be transformed into ETA- or PHI-expressions). A similar definition of LABEL becomes a little bit more involved.

Implementation

A preliminary version of evala, the evaluation function in LISP A, was coded in ordinary LISP for the PDP-6 computer. That version lacked some facilities that have been described here. Most important, it only permitted RHO-expressions to be evaluated for their side-effects, not for their value. On the other hand, there were some additional facilities in that system.

A modernized version of evala according to the specifications in this report is currently (March, 1968) available in LISP for the CD 3600 computer.

The crucial feature in these implementations was the use of a *queue* for expressions-to-be-evaluated. The need for this arises e.g. when we handle RHO-expressions that contain one or more occurrences of symbolic functions. Such a RHO-expression *is triggered* by the evaluation of an expression

(SETSUBSET ......);

it *itself triggers* evaluation of similar expressions (e.g., when SYMQUOTEd expressions are set as subsets of larger sets). Obviously this process may continue in a chain or tree. At each step, several new RHO-expressions may be triggered.

The order of evaluation of such expressions can be chosen in several ways:

1. *Depth-first.* If evaluation of expression e triggers expressions f1, f2, . . . , fk, we first evaluate f1 and all its consequences to the end of the tree; and only then start to evaluate f2.
2. *Queueing.* We keep a queue of expressions that are to be evaluated. If e triggers f1, f2, . . . , fk, these operations are put at the end of the queue, and the evaluation of all is postponed until the expressions before them in the queue have been handled. When we reach f1, we put the expressions that it triggers at the end of the queue; proceed to f2, etc.
3. *Queueing with priority.* This is like (2), except that expressions which are deemed particularly

significant, are permitted to step into the queue at some point other than its end.

Other alternatives, and more sophisticated ones, are also possible. In our implementation, we have preferred the "queueing with priority" scheme.

Other incremental computers

The System by Lombardi and Raphael.

The idea to use LISP as the basis for an incremental computer is not new; it was originally put forward by L. A. Lombardi and B. Raphael.[7] They describe a modified LISP system which can (in some sense) evaluate expressions, even when the values of some variables have not been specified.

Lombardi and Raphael specify three key requirements for an incremental computer. We shall relate the present work to theirs by discussing whether LISP A satisfies those requirements. The first of them is:

"The extent to which an expression is evaluated is controlled by the currently-available information context. The result of the evaluation is a new expression, open to accommodate new increments of pertinent information by simply evaluating again with a new information".

In LISP A, we can write expressions which satisfy this by using RHO-expressions. (We can also avoid paying the cost for it by using LAMBDA-expressions.) The "current information context" for forms with a RHO-expression as their leading function is information about subsets of the arguments in this form. We would say that the form has been completely evaluated when all one-member subsets of all arguments are explicitly known, and all combinations of them have been considered by the system. Usually, this is not the case, and evaluation is performed to an extent "controlled by the currently-available information context".

As we have seen, forms with RHO-expressions are stored away in such a way that they are "open to accommodate new increments of pertinent information" about subsets.

The LISP A system does not satisfy Lombardi and Raphael's second requirement ("algorithms, data, and the operation of the computer (!) should be specified by a common language") or their third condition (this language should be understandable by untrained humans). But neither does their incremental LISP, nor any other system we have heard of.

Our system is extremely inefficient in terms of computer time, and it can be assumed that theirs is less wasteful. On the other hand, LISP A seems to have the following two advantages over their system:

(1) When an expression is evaluated incompletely for lack of information, the system remembers this and resumes evaluation when further, pertinent information increments become available.

(2) Through the introduction of ambjects and symbolic functions, our system comes closer to having "a large, continuous, on-going, evolutinary data base", which should be the characteristic environment of an incremental computer. The data base of Raphael's program is identical to that of LISP 1.5, i.e. it is restricted to property-lists of atoms.

Lombardi has later published a more extensive treatise of incremental computers.[8] He there concentrates on the basic representation of data in core, and introduces his own system with three references in each cell. These seem to be quite different problems from the ones tackled in this paper, and a comparison is therefore not attempted.

Future Developments of LISP A.

The following developments seem natural:

A. Write a machine-coded version of evala.

B. Introduce a notation through which pseudo-parallell execution of several expressions can be performed. This is very natural, since e.g., the order of evaluation of the specializations of a RHO-expression is immaterial.

C. Attack storage problems by making use of backing storage, drum or disk. Facilities for parallel execution of expressions then become very important, because they help us to use the time when we are waiting for information from backing storage.

SUMMARY

LISP A is a modification and extension of LISP 1.5. Besides minor modifications, two new types of functions have been added to the language. One type *(symbolic functions)* is used to create and extend the data base. If ISFATHER is a symbolic function, evaluation of (ISFATHER JOHN DICK) will create a representation for the relation in the data base, without asserting its truth. This representation can then be used with conventional LISP functions, which set it true, ask whether it is true, etc. The other type (*RHO-expressions*) can be used to write a kind of rules of inference, which are automatically triggered in desired situations. The LISP A system is governed by such RHO-expression operators, which trigger each other. There is no coherent program, just a set of operators which communicate through the changes they cause in the data base. The paper gives a general description of the LISP A system.

ACKNOWLEDGMENTS

The author is indebted to professor John McCarthy at Stanford University for his kind guidance during the year 1966/67, when the work reported here was started. He would also like to thank fil.lic. Jacob Palme of the Research Institute of National Defense, Stockholm, Sweden, for many valuable discussions during the last year.

REFERENCES

1 J McCARTHY
Recursive functions of symbolic expressions and their evaluation by machine, part I
Communications of the ACM *3* (April 1960) p 184

2 J McCARTHY et al
Lisp 1.5 programmer's manual
The M.I.T. Press 1962

3 C WEISSMAN
LISP 1.5 primer
Dickenson Publ Co Belmont Cal 1967

4 E C BERKELEY et al
The programming language LISP: Its operation and applications
The M.I.T. Press 1966

5 D G BOBROW
Natural language input for a computer problem solving system
Doctoral Thesis M.I.T.

6 B RAPHAEL
SIR: A computer program for semantic information retrieval
Doctoral Thesis, M.I.T.

7 L A LOMBARDI B RAPHAEL
LISP as the language for an incremental computer in Ref. 4

8 L A LOMBARDI
Incremental computation
In Frank L Alt ed
Advances in Computers Vol 8
Academic Press New York 1967