Third edition
Aug 1972

The Programming Languages
LISP and TRAC
by W. L. van der Poel

Note from the editor (2008):
This handout is from 1972, and before using this document you should consider the following:

- The handout is about what is today an outdated form of LISP and is entirely based on LISP 1.5
- Compiled forms of LISP are not addressed at all
- There are so many good books published about LISP, Common LISP, SCHEME, EULISP and much more.

The author does not approve the distribution of this file because of these reasons.

This handout has been used for a Nato course in Copenhagen, in Japan as LISP course on a sabbatical in 1977 and of course in Delft.

Contents

In the last 15 years, there have been produced an overwhelming number of programming languages. The first languages made were called "problem oriented". A language such as FORTRAN could be called problem oriented for a large class of numerical problems, although the problems are not solved by the language, nor by any other language [1]. But a known algorithm can be more easily expressed and read by a human reader in a higher level language than in a machine language. Later many other languages followed. The most notable among these are:

ALGOL 60, an algorithmic language mainly meant for numerical calculations, but which has shown capabilities in other fields [2, 3]. Its design and rigourous definition has deeply influenced the whole field of programming languages.

LISP, a list processing language. The data structures in the form of linked lists and trees formed the basis of this language. Its forerunner was IPL V [4].

COBOL, a Common Business Oriented language in which large files of data and their description play the dominant role. The procedural side is rather verbose.

SNOBOL, a string manipulation language in which matching of substrings in strings are the main operations. Its forerunner was COMIT [5,6,7,8]

APL, a mathematical language in which the concept of vectors and matrices and the implicit operations on all elements of vectors is predominant. Its notation of programs is the shortest among all programming languages, although this does not always mean that the algorithms are equally perspicuous [9,10,11].

FORMAC, a language to manipulate with symbolic formulae. It is based on FORTRAN. Later, many more formula manipulation systems followed, of which only MATHLAB will be mentioned [12,13].

ALGOL 68, one of the "big" languages, oriented toward a big class of problems, numerical as well as symbolic or of the list-processing kind [14,15,16,17]

In the sketch above, I have not tried to be in any way complete. I have not mentioned the class of simulation languages such as CPSS or SIMULA 67 [18,19], nor list manipulation languages such as SLIP [20]. The class of simplified languages for conversational use of which BASIC [21] has become the exponent could not be missed. Other "big" languages such as PL/I deserve mentioning [22]. A special class of languages are the system implementation languages for writing systems software. They must be able to express all

facilities of a particular machine (e.g. PL/360 [23], BLISS [24]) or better still to enable systems programming to be written in a machine free way, so-called portable software which can be transported from machine to machine with a minimum of effort. Examples of such languages are AED 0 [25], BCPL [26] and the SIMCMP/STAGE2 [27] system. A great many other languages are strongly inspired by the ALGOL ideas, whatever this may mean. Examples are ALGOL W [28], PASCAL [29] and POP2 [30]. All three make an attempt to cover a large range of problems; the last one combines conversational properties on a time-sharing network with some very advanced features such as list processing and partial parametrization of function calls. As the last few languages, which I should like to name explicitly are TRAC [31], a string manipulation language for conversational use of a singularly elegant and basic construction, and EULER [32], a language where syntax and semantics were defined in a closely linked way resulting in a language very formally described, powerful and easily implementable.

The purpose of these lectures is to give an approach to some basic aspects underlying many programming languages. For these aspects I have chosen data structures, in particular linked list structures, the semantic way of defining programming languages, and the fundamental concepts of the name/value couple (or content function), the function call and parameter mechanisms. For demonstrating these concepts I shall make use of two particularly simple languages, both used in a conversational way: LISP and TRAC. These languages were chosen for several reasons. LISP is the prototype of a list processing language. It can be defined in terms of itself. It is easily extensible and the principle of small compilers can be demonstrated in it. TRAC has the utter simplicity of having only a single mechanism for name evaluation as well as function call and go to. And last but not least, these languages are small enough to be implemented and demonstrated on a 4K word mini-computer.

In the course of the story many references will be made to other languages to show the equivalence or similarity between mechanisms of the languages. As ALGOL 68 will be treated very extensively in another lecture, this language will often be mentioned for comparison.

As these writings are an incomplete rendering of a story that will both be told and shown directly on a computer, more stress will be laid on completeness for the reader of the examples, than on the text filling the gaps.

1. The Programming Language LISP

LISP is one of those languages, which has been essentially created by one man, John McCarthy. After its predecessor IPL V (Information Processing Language number five) which was developed to simplify the expression of heuristic programs, LISP came as a complete language in 1960. IPL V was still very close to a typical machine language and consisted of sequences of single instruction, LISP suddenly came with the concept of functions (and functions only!) together with a computerized form of lambda calculus. Although there has never been an attempt to bring LISP under the aegis of an organization or to standardize it in any way, it has lived a very stable life. Apart from minor differences in implementations which can mostly be overcome by its extensible nature, the elementary functions are the same for all implementations including input and output.

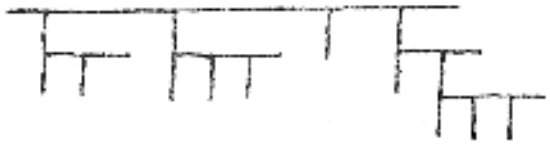
The data structures needed for heuristic programs, applications in the field of artificial intelligence and mechanized proof-checkers had to have the property that they could grow and shrink in many direction. The concept of lists was very suitable for this purpose. In fact, the tree structures have been capitalized upon so much in LISP, that it is the only data structure available. There are no arrays, linearly ordered and of elements all of the same kind as in ALGOL. Even the programs in LISP are fully expressed in the same data structure. This is one of the powers but at the same time one of the weaknesses of the language. Keeping the programs in their original form will necessitate the system to be of an interpretive nature, hence rather slow. Compiling into machine code will make LISP lose its ability of being self-extensible unless the source text is kept alongside the compiled code. A change in source text will necessitate a recompilation and of course in this language the compiler itself is callable as a function at run time. We shall deal here only with the interpretive version.

Although well stabilized by a kind of folklore, there are very few books treating the theory of LISP. There is The LISP Programmer's Manual but this is more an implementation bound manual for users; there is also the Berkeley and Bobrow book, an encyclopaedic compilation of material but hardly a textbook and there is only one real textbook by C. Weissman. [33, 34, 35, 36]

1.1. List structures

All data in LISP are represented by trees. Although it should be thought that trees with an arbitrary number of branches in every node e.g.

is a more general structure than a tree where every node has only two branches



one can easily see that the pictures above are essentially representing the same tree. The horizontal branches are used for further elements on the same node, the vertical branches are used for the next level. The end points in the horizontal direction are usually marked by an object called NIL and usually represented in most implementations by a pointer to 0. This object NIL is a primitive notation of the language and can be regarded as the empty list. Syntactically a list can now be described by:

list : atom ;
left parenthesis symbol , list , dot symbol , list , right parenthesis symbol .

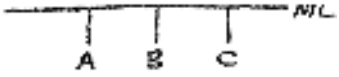
atom : character ;
atom, character .

character : other symbol

The usual representations are

left parenthesis symbol	(
right parenthesis symbol	)
dot symbol	.

the other symbols comprise all characters available except the three above. E.g. a list consisting of thee atoms A, B and C



could be represented by (A.(B.(C.NIL)))

This is so clumsy that immediately an abbreviation shall be invented for this i.e. (A B C) , where the space now is a fourth syntactic mark serving to separate the atoms. The size of the space is of no importance. Also the

separate the atoms. The size of the space is of no importance. Also the change to a new line is regarded as a space.

One could describe the rules for deriving list notation from dot notation as follows:

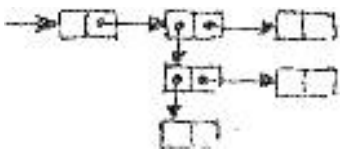
Rule 1) A dot followed by a left parenthesis together with the corresponding right parenthesis may be struck. (By definition the parenthesis are always balanced and the balance will be maintained by this rule).

Rule 2) A dot followed by NIL may be struck.

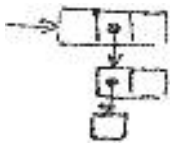
A more complicated structure such as ((A B).(C D)) could be represented without the dot as ((A B) C D)

Unfortunately the dot can not always be eliminated, but we shall see that its actual use is very restricted and all practical programs will not show any dot.

Very often these list structures are implemented in the usual linearly arranged kind of stores we have on the present day computers in the following way



Every bifurcation is represented by two adresses, point to the "downward" and to the "rightward" branches respectively. The actual addresses used can be completely arbitrary. The nodes could be completely scattered in store. This is one of the great advantages of LISP. One is relieved from the allocation problem but a new cell can be taken anywhere where a free cell is available. The way to represent a node by two addresses is only a choice of the implementer. One could equally well choose the downward branch to be represented by a pointer but the rightward branch being the next physical location.



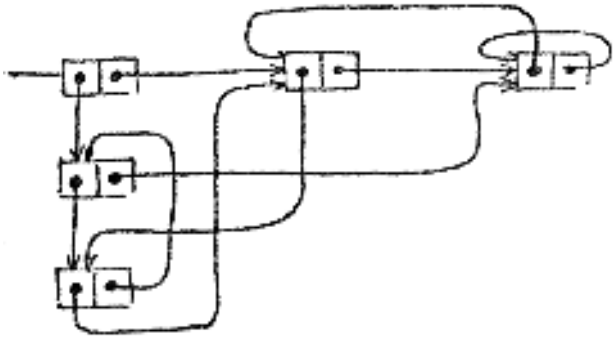
We shall not deal with any of these implementation dependant details as they are completely irrelevant to the language.

Trees can be knotted or circular. I.e. an element of a list could be a node on the same, a higher or lower level. These knotted trees can be depicted very easily but they cannot be represented so easily in the syntactic symbolism. Whether (A A) is a list of two elements both referring to the same atom A or of two elements each referring to a different atom with the same identifier cannot be seen from the outside. (We have made an implementation in which this difference can be shown in printing).

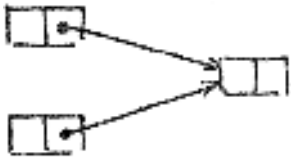
As most LISP systems have adopted the convention that every atom with a name differing from other names, will only exist once. The same instance of that atom will be shared by all lists using that identifier. But this is inherent in the system built on top of the primitives and the convention can be broken. One could compare the uniqueness of identifiers with that in FORTRAN (without subroutines). We shall show how also a local use can be made of the same identifier by having more than one atom with the same identification.

On the other hand, numbers will always be attached to a newly created atom. Every intermediate result, e.g. the number 5 will be a new instance, i.e. a new atom 5. The reason is clear. When the same 5 would be used in a shared fashion, changing this 5 to 6 would change all other values 5 in the whole program to 6.

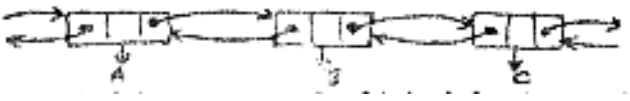
A knotted tree



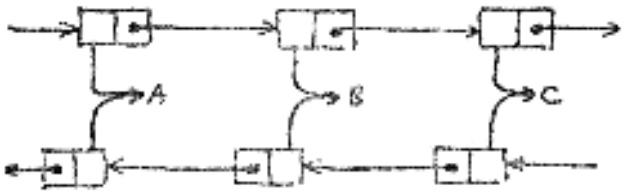
A shared sublist



There are list systems where the linkage is not strictly one sided as is postulated here. But the doubly linked list



can also be represented by two singly linked lists as follows.



1.2. Atoms

To distinguish the atoms from lists usually a specially marked cell is used. The downward pointer will be regarded as carrying this atom mark and will point to the sequence of character representing the atom. The rightward pointer will be normal connexion point for a so-called "property list", containing properties attached to the atom. If there are no properties, the list is NIL. The sequence of characters in the atom can be regarded as its identifier, although any sequence of characters will act here as an identifier.

A special case is formed by numbers. We shall restrict ourselves to signed integers only. They are also represented by an atom with the atom mark, but they do not have the number referred to by the downward pointer. Instead the number is stored as the only property of that atom and the downward pointer is NIL (plus the atom mark of course). For those who like pictures:

Again these pictures are very implementation dependent and we shall not make any further use of them.

Numbers are always standing for themselves, that is, the value of a number is always that number. E.g. the value of 3 is 3. The only property of 3 is being 3.

1.3. The notation of functions

The usual notation of a function in mathematics is F(X) denoting the function with identification F applied on the argument X. By the last we always mean the value of X and not the letter X itself. In LISP two notations are in use called resp. inner LISP and outer LISP.
In inner LISP the notation will be (F X) or (F X Y ..) for more arguments. The first atom of the list will always be the function identification, the following atoms stand for the value of the arguments.
As on the outer level very often the arguments stand for themselves, the notation of outer LISP will be followed. This is again F(X) . But now the argument X stands for itself, i.e. the letter X and will not be evaluated.

If in inside LISP an object, representing itself literally, is needed one can provide this by using the function QUOTE.
Hence (QUOTE X) delivers X.

What can be written as FN(X) in outside LISP could also be written as (FN(QUOTE X)) in inside LISP. The main reason of using outside LISP is alleviating the user to write QUOTE all over the place.

It will appear later that QUOTE is not to be regarded as a primitive of the language, but can be defined in terms of other functions. In fact only about 12 object need to be regarded as primitive in the sense that they cannot be defined in terms of other primitives. They are NIL, CAR, CDR, CONS, EQ, ATOM, NUMBER, RPLACA, RPLACD, COND, READ, PRINT. The last two could be built up from INCHAR and OUTCHAR, but for the moment we shall regard them as primitive.

1.4. The primitive functions

Let us assume for the moment that X has the value (A B C) somehow. We shall later see how a value is attached to a variable.

The primitive function CAR will take the first element from the list supplied by its argument. Hence (CAR X) gives A.
The name is a traditional one, standing for Content of Address Register. It can be changed later.

The primitive function CDR will give the remainder of a list when the first element is taken away. Hence (CDF X) gives (B C).
The name is again a traditional one, standing for Content of Decrement Register. This has to do with one of the first implementations of LISP on the IBM 7094.

The CAR and CDR functions are the most primitive ones. In the picture of the cells with a downward and a rightward pointer CAR takes the downward direction and CDR the rightward direction in scanning a tree. One can also say that CAR and CDR are the functions of taking the content of the left half and the right half of a cell respectively.

CAR and CDR do not destroy any existing structure. They only take a look. In combination they can be used to access any element of a list.
E.g. (CAR(DR X)) will give B and (CAR(DR(CDR X))) will give C.
In fact these combinations are so frequent that a set of combined functions is built right into the system. E.g.
(CADR X) will have the same affect as (CAR(CDR X))

From now on we shall assume that all functions of the form C...R with an arbitrary number of A's and D's in between will present in the system.

Two special cases deserve attention. Taking CAR of an atom will be signalled as an error. E.g. (CAR(QUOTE X)) is wrong. It would mean that through the atom mark one would get access to the character string, forming the identification of the atom. But this part of the object is no more a LISP object and disaster could follow.

The other case is taking CDR of an atom. This is allowed and will reach the property list of the atom, about which later. A special case of this is (CDR NIL) which will give NIL.

The next very important function is CONS. This is a function of two arguments. It CONStructs a new cell of which the CAR and CDR part are made equal to the given arguments. Examples:

(CONS 3 4) gives (3.4)

(CONS X X) gives ((A B C)A B C) The dot is eliminated!

(CONS(QUOTE X)(QUOTE Y)) gives (X.Y) and so does CONS(X Y) in outside LISP.

The CONS function creates a new cell by taking it from a pool of free cells. This pool is in essence the same as the heap in ALGOL 68. Whereas LISP is a type-less language which needs a single cell for an atom or for a reference to another list, in ALGOL 68 a generator must claim an appropriate chunk from the free space to accomodate the required mode. In languages such as FORTRAN, all working space is assigned in a static way at compile time (except for a very temporary stack for intermediate results of expressions). But ALGOL 60 had some form of generator hidden in the local declarations. They create a new place for the type required dynamically. In ALGOL 60 this regime could be strictly played on a stack as the appearance and disappearance is always in strict LIFO order (Last In First Out) except for _own .

If all free space is used up, the system performs a so-called garbage collection. All cell which still can be reached are marked, all unmarked cells are joined to a new set of free cells and the marking on the still active cells is removed. The system with garbage collection has always been opposed to another system used in SLIP where each list keeps count of the number of references being made to it in a reference counter. However, this system breaks down on list structures which can be knotted or circular. Later versions of SLIP have been changed and also have adopted garbage collection.[43] If one stays away from running close to the capacity of the store, garbage collection does not consume a large proportion of processing time (<10 percent) but when running close to the limit van be very bad.

The possibilities of ALGOL 68 allow the definition of the basic primitives of LISP in the following way:

```
struct l = (ref al car, cdr);       ¢ a list is a pair of references to
                                      elements of mode al ¢
union al = (a , l);                 ¢ the mode al is united from atom and list ¢
struct a = (string n, ref al cdr);  ¢ mode atom has a name and a reference to
                                      a property list ¢
proc CAR = (ref l x)ref al : car of x;
                                    ¢ CAR selects the first field ¢
proc CDR = (reg al x)reg al : cdr of x;
                                    ¢ CDR selects the second field ¢
proc CONS = (ref al x, y)ref l : l := (x,y);
                                    ¢ CONS generates a new element of mode
                                    ref l ¢
proc ATOM = (ref al x)ref a : (ref a :: x ¢ TRUE ¢ NIL);
                                    ¢ test for identity of references. Different
                                      instances of numeric atoms are different
                                      according to this definition ¢
proc RPLACA = (ref l x, ref al y)ref al : car of x := y;
proc RPLACD = (ref al x, y)ref al : cdr of x := y;
                                    ¢ The basic assignment functions ¢
```

These declarations are doing more than corresponding LISP functions as the ALGOL 68 parameter mechanism is involved. Furthermore it is fair to say that these things were not invented in ALGOL but in LISP. The selection of ALGOL models the contents of function, and the generator models the CONS.

The function CONS and the functions CAR and CDR are in some way complementary, i.e.
(CONS(CAR X)(CDR X)) will give X as result. But one has to be careful! The result is equal to X but by no means identical to the original object X as a new first cell has been CONSed. The tail, however,is still shared with the old X.

A few more primitives are needed. The next one is the function ATOM of one argument. It gives the value T (for true) if the value of the argument is an atom, otherwise it gives NIL. NIL doubles for false. This could be regarded as a flaw in the language, but it is a generally adopted and very convenient convention. The object T is not primitive. It could be regarded as an object having T as its permanent value. Later its explicit definition will be given. Examples:
(ATOM X) gives NIL (X still stands for (A B C) as in the previous examples)
(ATOM(QUOTE X)) gives T. The atom X itself is of course atomic.
(ATOM NIL) gives T. An empty list is not a list but an atom!

A special kind of atom test is NUMBER. In literature this function is generally called NUMBERP but on the demonstration machine all standard objects are shortened to 6 letters. NUMBER is a function of one argument. It is true if the argument is a number otherwise it gives NIL. Examples:
(NUMBER X) gives NIL . X is a list, not even an atom
(NUMBER(QUOTE X)) gives NIL . X is a name, not a number
(NUMBER 3) gives T

The last primitive function is EQ of two arguments.
It gives T if the value of the arguments are identical, i.e. the same object Examples: (EQ(QUOTE A)(QUOTE A)) is T. There is only a single instance of the atom A itself.
(EQ X (CONS(CAR X)(CDR X))) gives NIL. The newly constructed value is perhaps equal to X but it is not the same instance.
(EQ 2 2) gives T . One would have expected NIL because every number is a new instance of that value. Logically speaking this is true. Only tradition has forced us to make the exception for numbers.

1.5. Conditional expressions

The construct to make use of the logical functions is the conditional expression. It is written as a function, but the number of arguments in indeterminate. Each argument is a list of two elements. E.g.:

(COND(A B)(C D)(E F)(T G))

The value of this expression is determined in the following way:
If A has a value which is not NIL, then the value of the whole conditional expression is B, else if A has the value NIL the next pair is considered and treated in the same way as the first pair. In practically every other language a similar construction exists. E.g. in ALGOL 68 the action could be compared with:

if A then B else if C then D else if E then F else G fi fi fi

The fact that the last pair begins with T makes that,if no other pair has succeeded in producing a value, the last pair will in any case. It is an error to "fall through the bottom" of a conditional expression and will be signalled as such. It is important to note that it is not required that the first of a pair is T. Being not NIL is sufficient. This is a very useful property.

Conditional expressions should not be regarded as functions. One could imagine that a new function IF could be defined with 5 arguments so that (IF A THEN B ELSE C) would stand for (COND(A B)(T C)). The big difference is that in function call all arguments are evaluated before the function is applied to the evaluated arguments. This is exactly what we do not want in a conditional expression. If A is true, then the value is B and C has never been evaluated, but if A is NIL then B will never be evaluated.

We have not yet seen how new function can be defined but we could now write a conditional expression having the same action as the logical AND. E.g. (AND X Y) could be written (COND(X Y)(T NIL)) . Only if both X and Y are true the value T will be delivered. In the same way (OR X Y) could be written (COND(X T)(T Y)) . (NOT X) could be written (COND X NIL)(T T)) but (EQ X NIL) is shorter an has the same effect. The same function NOT can also be used to test whether a list is empty. It is then usually called NULL.

1.6. Lambda expressions

To be able to define new functions we need a way if indicating which are the variables to be used as parameters and in which order. The mechanism for this is the lambda notation, introduced by Church. If we write in mathematics f(x,y) = x - y then the variables x and y are called bound variables. They are used for descriptional purposes only. We could have written as well f(u,v) = u - v . But f(u,v) = v - u is a different function! Now describing a function body and giving it a name are two completely different things. The so-called formal parameters do not belong to f but to the expression after the = sign. LISP has adopted a list notation for the lambda forms in the following way: (LAMBDA (list of formals)(function body)) Now we can write more correctly for the AND function:

(LAMBDA(X Y)(COND(X Y)(T NIL)))

Lambda form can be used in the same positions as function names LAMBDA(X Y)(COND(X Y)(T NIL))) (T T) will give T in the same way as in outer LISP AND(T T) would have done.

1.7. The association list

Inside the system the mechanism to bind the actual parameters to the corresponding formal parameters is the association list. During the evaluation of a lambda form the actual parameters are bound to the formal parameters in a list of dotted pairs. E.g. in the case above the association list reads as follows ((X.T)(Y.T)) . The association list could carry many more associations of outside calls made earlier. It works in a stack like fashion. The last value associated is found first. New associations are added on the left. When the evaluation is ready, the association list is restored as it was before entering the function and older associations will now be found.

The system as described above is the classical, interpreted form of LISP. More modern implementations keep a special cell reserved on every atom, the so-called value cell, where the last value associated with the name is kept. Previously associated values are temporarily put on a single stack when a function is entered and the newly associated value of the formal parameter is put in the value cell. As the calling and completion of all functions is proceeding in a strictly nested fashion (there are no jumps yet!) the previously associated values can be restored at the end in exactly the reverse order in which they were placed on the stack.

The kind of binding that takes place on the association list is best comparable with the relation "to possess" from ALGOL 68 as performed in elaborating a call. The original routine denotation with its formal parameters pack possesses a routine which begins with identity declarations for each formal parameter. On the call the skips are replaced by the actual parameters, the actual parameters are elaborated and the formal identifier is made to possesses the value of the actual. In exactly the same way the association list lists the values possessed by the formals in the form of pairs.

There are slight differences. In ALGOL one cannot "quote" an argument on the actual side. And in LISP there is hardly a difference between the external object and the internal object. A program in LISP is actually kept in the same form; the parentheses are replaced by the structure internally but they are reinstated when a form is printed.

In LISP the associations are only kept during the evaluation of a lambda form. This coincides with the dynamic scope of the formals in ALGOL in a closed clause.

1.8. Property lists

In LISP there is another method to attach properties or values to any atom. Each atom has its CDR pointer available as its connexion point to a property list. On this property list the properties are inserted as doublets, couples of two list items. The first is a so-called "indicator", giving the kind of property, the second item is the substance . We shall only introduce three kinds of properties here.

The indicator APVAL indicates that the atom has a permanent value which follows in the second item.

The indicator EXPR indicates that the atom is a normal function. The lambda form describing the action follows in the next item.

The indicator FEXPR indicates that the atom is a special kind of function. A somewhat special lambda form follows to describe its action.

For example the X we have used in our first examples was suppose to have the value (A B C). This simple meant that we had attached to the atom X the property list (APVAL (A B C))

In the same way when AND is the name of the logical "and" function then we have to attach the property list (EXPR(LAMBDA(X Y)(COND(X Y)(T NIL)))) to the atom AND.

A single atom can have more than one property attached to it. E.g. we shall assume that the atom T has the following property list:

(APVAL T EXPR (LAMBDA(X)X))

i.e. use T as a value, its value will be T again, but applied as a function to an argument it will deliver that argument. We shall later see how handy this will be.

1.9. The function DEFLIS

To be able to attach property lists to atoms there is a function DEFLIS (called DEFLIST in literature) which will do just this. As it is mostly used to attach literally a piece of text to literally a particular atom, the arguments will practically always have to be QUOTEd. Hence it is simpler to do a DEFLIS right on the outside level. The form in which DEFLIS is used is a bit clumsy. It is:

DEFLIS(((name1 property1)(name2 property2)...(namesx propertyx))indicator)

So there are two arguments. The second is the required indicator, the first is a list of an arbitrary number of pairs. The property1 will be attached to the atom name1 etc. all with the same given indicator.

The first thing we shall do is making a new function to define normal functions with the indicator EXPR

DEFLIS(((DEFINE(LAMBDA(L)(DEFLIS L EXPR))EXPR))))

Even this function is too clumsy for us. For the moment we shall not define lots of functions at the same time but one at a time. At the same time we want to save us the trouble of writing LAMBDA. Hence we make

```
DEFINE(((EX(LAMBDA(X Y Z)(DEFINE(CONS(CONS X(CONS(CONS LAMBDA(CONS Y
    (CONS Z NIL))NIL))NIL))NIL))))))
```

This looks formidable but enables us to define a function with

EX(name(list of formals)(body))

E.g. EX(AND(X Y)(COND(X Y)(T NIL))) will define the function AND

The great number of CONSes in the definition of EX could have been prevented by making a function for composing lists. To pack one element as a list one can write (CONS 5 NIL). This makes (5) out of 5. A list of two elements can be made with (CONS 1(CONS 2 NIL)) producing (1 2) etc. Although we are not able yet to define functions with an arbitrary number of arguments, we shall give the function LIST which just does this. It makes a list out of an arbitrary number of arguments. With the help of LIST the function EX could have been defined as:

```
DEFINE(((EX(LAMBDA(X Y Z)(DEFINE(LIST(LIST X(LIST LAMBDA Y Z ))))))))
```

In the LISP system used here for demonstration, the function DEFLIS and hence also DEFINE and EX have the property that even standard built-in function can be redefined. E.g. one could write:

```
EX(CAR(X)(CAR X))
```

A new function CAR is now defined, but during the reading of the text of this line the old function is still in force. Hence the new CAR is defined in terms of the old one. If one would again redefine `EX(CAR(X)(CAR X))` one would get a circular definition and the execution time becomes infinite.

It is also possible now to give other names to the standard functions e.g.:

```
EX(HEAD(X)(CAR X))  EX(TAIL(X)(CDR X))  EX(NEWCELL(X Y)(CONS X Y)) etc.
```

1.10. Examples in the simple system

We have now available the objects and functions:
NIL, CAR, CDR, CONS, ATOM, NUMBER, EQ, QUOTE, COND, LAMBDA, DEFLIS, DEFINE, EX
and we shall show how further functions can be defined.

```
EX(NULL(X)(EQ X NIL))
```

The function NULL will test whether a list is empty or not. Because false and empty are both represented by NIL, the same function NULL can be used as NOT. Although we now define these functions in term of more primitive functions, many are built in for efficiency reasons and we shall say so if this is the case. NULL is built in. The explicit redefinition makes the built-in function inaccessible.

```
EX(EQUAL(X Y)
  (COND((EQ X Y)T)
       ((ATOM X)NIL)
       ((ATOM Y)NIL)
       ((EQUAL(CAR X)(CAR Y))(EQUAL(CDR X)(CDR Y)))
       (T NIL) ))
```

If X and Y are identical, they are certainly equal. Now atoms which are equal are also identical, hence if X is an atom and if it would have been identical with Y, the first EQ would have found it so if X is not identical to Y and X or Y is an atom, then the result is certainly NIL. The fourth test is the case when both arguments are list. Here the major feature of recursion comes in. If now the first elements of the lists are equal, then the result depends on the equality of the remainder of the lists. The function EQUAL is used twice in a recursive manner. If even this is not the case, then the result is certainly false. EQUAL is a built-in function.

Another example of recursion is the function APPEND to join two list to a single one.

```
EX(APPEND(X Y)
  (COND((NULL X)Y)
       (T(CONS(CAR X)(APPEND(CDR X)Y))) ) )
```

The result of APPEND((A B C) (D E F)) would be (A B C D E F). This function is by no means efficient, as the first list has to be broken down, before the result is built up. The function NULL is always used when testing on the end of a single level list must be done.

A function to reverse the elements of a list on the main level present some peculiar difficulties. A first solution would be:

```
EX(REVERSE(X)(COND
       ((NULL X)NIL)
       (T(APPEND(REVERSE(CDR X))(CONS(CAR X)NIL))) ))
```

But this function is quadratically inefficient for long list. REVERSE as well as APPEND has to run through all the remaining elements. A better solution would be to take the elements off the given list on by one and CONS them onto a result list. The trouble is that we have no "hands" to hold that result. There is not yet any assignment yet! The solution is to have the work done by an auxiliary function with two parameters. The second parameter can be used as the second hand to hold the temporary result.

```
EX(REVERSE(X)(AUX X NIL))
```

Reverse only calls AUX with an empty second parameter.

```
EX(AUX(X Y)(COND
       ((NULL X)Y)
       (T(AUX(CDR X)(CONS(CAR X)Y))) ))
```

Now Y is the temporary and final result.

The above example has shown that it is possible to build up a language theory without the concept of assignment and without the usual program sequencing and the unavoidable go to's. The only concept used so far is function calling together with the binding mechanism for parameters. But this is not always as efficient as it is elegant and we shall give a more efficient solution to REVERSE later on.

REVERSE((A B (B D) E F)) would give (F E (C D) B A) as result. It only reverses on the main level. A function reversing on all levels would run:

```
EX(SUPERREVERSE(X)(COND
       ((ATOM X)X)
       ((NULL X)NIL)
       (T(APPEND(SUPERREVERSE(CDR X))(CONS(SUPERREVERSE(CAR X))NIL))) ))
```

SUPERREVERSE((A B(C D)E)) would give (E(D C)B A)

Remark that now the test for end is an ATOM test. The reversing stops when an atom is reached. The CONS is needed for packing the second argument of APPEND as a list.

Another example where the atom test signals the end is a function to flatten a list to a single level.

```
EX(FLATTEN(X)(COND
       ((ATOM X)X)
       ((ATOM(CAR X))(APPEND(CONS(CAR X)NIL)(FLATTEN(CDR X))))
       (T(APPEND(FLATTEN(CAR X)(FLATTEN(CDR X))))) ))
```

FLATTEN((A B (C (D E))(F G) H)) would give (A B C D E F G H)
It is left to the reader to inspect the special difficulty of this function.

Two more examples in the same style are LENGTH (on a single level) and TOTALLENGTH (all levels plus the nodes). This requires the use of PLUS, which is almost self explanatory.

```
EX(LENGTH(X)(COND((NULL X)0)(T(PLUS 1(LENGTH(CDR X))))))
```

An empty list has length 0, otherwise it is 1 longer than its tail.

```
EX(TOTALLENGTH(X)(COND((ATOM X)1)
        (T(PLUS(TOTALLENGTH(CAR X))(TOTALLENGTH(CDR X)))) ))
```

A beautiful example of the use of recursion is the problem of the towers of Hanoi. On a board there are three pegs. On peg 1 there is initially a stack of pierced disks in such a way that the disk at the bottom is largest, and that the diameter of each successive disc, going to the top, is smaller than the preceding one. The problem is to move all disks to peg 2 by moving only one disk at a time and with the additional restriction that never a larger disk shall be above a smaller disk. Peg 3 may be used for shunting. The problem is soved recursively by remarking that for one disk the problem is solved by transferring this disk directly. So we only have to solve the problem of temporarily moving n-1 disks from peg 1 to peg 3, then move the bottom disk to peg 2 and then moving back the n-1 disks from peg 3 to peg 2.

```
EX(HANOI(X Y Z N)(COND((EQ N 1)(CONS X Y))
            (T(LIST(HANOI X Z Y(MINUS N 1))
                   (CONS X Y)
                   (HANOI Z Y X(MINUS N 1)) )) ))
```

Recursion must be handled with care. This is shown in the next example of a term from the series of Fibonacci:

```
EX(FIB(X)(COND((EQ X 0)0)
              ((EQ X 1)1)
              (T(PLUS(FIB(MINUS X 1))(FIB(MINUS X 2)))) ))
```

Here the recursion is only used for doing the counting. We see how near this solution comes to the iterative solution. In fact, iteration can be transformed into recursion an vice versa. We shall revisit the problem in chapter 1.19.

As a last example we shall give the function of Ackermann, one of the simplest non-primitive recursive functions.

```
EX(ACK(M N)(COND
            ((EQ M 0)(PLUS N 1))
            ((EQ N 0)(ACK(MINUS M 1)1))
            (T(ACK(MINUS M 1)(ACK M(MINUS N 1)))) ))
```

This follows closely the mathematical definition. It is beautiful to trace this function and follow its tremendous complecation of recursion levels. See Appendix 2 for an actual example.

McCartyh's 91-function is trivial in this language

```
EX(F91(X) (COND((LESSP 100 X)(MINUX X 10))
               (T(F91(F91(PLUS X 11)))) ))
```

It is trivial to get proper results, it is not trivial to prove its equivalence with

```
EX(F91(X) (COND((LESSP 100 X)(MINUS X 10)) (T 91) ))
```

And the language as such does not help to provide a proof. For a proof see [45].

1.11. Addition

Although we have occasionally used the function PLUS, we shall show in this chapter, that PLUS is by no means a primitive. It can be completely defined in terms of CAR, CDR, CONS, EQ and COND and nothing else. We shall assume that numbers will now be represented by a list of their digits. It is easier to process from left to right by CAR but addition has to be done from right to left. Therefore we shall define

```
EX(SUM X Y)(REVERSE(SUMREV(REVERSE X)(REVERSE Y)0)))
```

First we shall define a kind of successor function

```
EX(SUCC(N)(COND((EQ N 0)1) ((EQ N 1)2) ((EQ N 2)3) ((EQ N 3)4) ((EQ N 4)5)
            ((EQ N 5)6) ((EQ N 6)7) ((EQ N 7)8) ((EQ N 8)9) ((EQ N 9)0) ))
```

This is a simple enumeration of all possibilities.
The predecessor is definable in terms of the SUCC in two steps

```
EX(PRE(N)(PRE2 N 0))
EX(PRE2(N M)(COND((EQ(SUCC M)N)M)
            (T(PRE2 N(SUCC M))) ))
```

It is by no means the most efficient method!
The next function will determine the direct sum digit of two digits, disregarding the carry.

```
EX(DIR(A B)(COND((EQ A 0)B)  (T(DIR(PRE A)(SUCC B))) ))
```

This corresponds with if a - 9 then b else a- + b'
In the same way the carry can be defined

```
EX(CARRY(A B)(COND((EQ A 0)0) ((EQ (SUCC B)0) 1) (T(CARRY(RE A)(SUCC B))) ))
```

In a complete addition we have to cater for three terms, the two digits to be added and the carry from the previous position. Hence

```
EX(SUM3(A B C)(DIR(DIR A B)C))
EX(CARRY3(A B C)(COND((EQ(CARRY A B)1)1)
            (T(CARRY(DIR A B)C)) ))
```

Use has been made of the fact that the sum can never exceed 19. Now SUMREV can be defined. It has three arguments, two lists of digits representing the reversed numbers to be added and the carry, made 0 on the initial call (see SUM). The first few lines cater for the case that the list are exhausted or that one list is shorter than the other. Then the shorter one is extended to the length of the other.

```
EX(SUMREV(U V P)(COND
  ((EQ U NIL)(COND
    ((EQ V NIL)(COND
      ((EQ P 0)NIL)
      (T(CONS 1 NIL))))
    (T(COND
      ((EQ P 0)V)
      (T(SUMREV(CONS 0 NIL)V 1))))))
  ((EQ V NIL)(COND
    ((EQ P 0)U)
    (T(SUMREV U(CONS 0 NIL)1))))
  (T(CONS(SUM3(CAR U)(CAR V)P)(SUMREV(CDR U)(CDR V)(CARRY3(CAR U)(CAR V)P)))) ))
```

Now SUM((1 2 3 4 5 6 7 3 4 5 3 4)
         (5 6 7 8 9 0 9 8 7 8 7 4)) would give
         (6 9 1 3 4 7 7 2 2 4 0 8)

The length of the numbers is irrelevant except for the capacity of the machine. Even the number base can be changed very easily. E.g. the same can be done in the ternary system by redefining

```
EX(SUCC(N)(COND((EQ N 0)1) ((EQ N 1)2) ((EQ N 2)0) ))
```

All other functions remain the same.

But from now on we shall work with the functions PLUS and MINUS as built into the system. The function PLUS can even accept an arbitrary number of arguments. We shall soon learn how to define functions with an arbitrary number of arguments, a feature not present in most other programming languages

1.12. The functions EVAL and APPLY

We have spoken several times about the evaluation of a variable. Two different mechanisms have been discussed to bind a value to a variable. One is a dynamic binding on the association list, the other was a semi-permanent binding by means of an APVAL property on the property list. In inner LISP the arguments of functions were always evaluated before the function (or the lambda form attached) is applied to them. The evaluation of variables and the application of a function can be done separately by two functions EVAL and APPLY. In fact these two functions are the heart of the whole system. In a further paragraph we shall formally define them. For the moment we only want to know them so that they can be used.

EVAL is a function of two arguments. The first is the form to be evaluated, the second is the initial association list. Examples:
EVAL(A ((A.3))) gives 3
If B has been given the permanent value (P Q R) by
DEFLIS(((B (P Q R)))APVAL) then
EVAL(B NIL) will give (P Q R)
The classical LISP systems will give the value bound on the property list before searching the association list. In the present system we have reversed this order. In that case the property value could be regarded as being the bottom value in the stack of values. In the more modern implementation with a value cell on each atom there is no need for a distinction any more. EVAL(B ((B.5))) will give 5 although the property list of B as shown by executing CDR(B) will still show (APVAL(P Q R))

APPLY is a function of three arguments. The first argument is a function, given as the name of the function or directly as a lambda-form; the second argument is a list of arguments to which the function mentioned in the first argument must be applied. This list of arguments will be taken as it stands. They could have been evaluated previously by another function or it is the intention of not having them evaluated at all. The third argument of APPLY is again the initial association list, often NIL at the beginning. If APPLY is used in inner LISP the second argument of APPLY is of course evaluated but the resulting value is a list of arguments for the function to be applied and this list of arguments is not evaluated any further. In outer LISP this distinction is not made as all arguments are QUOTEd in any case. Example:
APPLY(CAR (P Q R) NIL) will give P as CAR is applied to (P Q R)
With the above definition of B however
(APPLY CAR B NIL) will also give P and
(APPLY CAR B ((B X Y Z))) will give X.

In many implementations the functions EVAL and APPLY are combined to a single function. In fact the only difference is that EVAL evaluates the arguments and applies the function in (FN X) and APPLY only does the application of the function and takes the arguments as they stand.

1.13 Input and output

Until so far we have only made internal machine programs without input from the outside world or output to the outside world, except for the programs which was typed in and the result delivered. This was implicitly done by the system. We shall now bring this into the open.

Output is done by the function PRINT of a single argument. The value of the argument is printed (or the argument itself in outer LISP) and the function also returns with that value as its result. As all objects are lists, PRINT need only be able to print a single list (special case: an atom). E.g. PRINT(A) will print an A. Or in inner LISP when X has the value (P Q R) (PRINT X) will print (P Q R). If in inner LISP a single character must be printed we have to write (PRINT(QUOTE X)) . This will print the letter X only.

To be able to print characters which normally would have a syntactic meaning such as space, dot or parentheses, there is a device for super quoting these characters and making them to ordinary characters. The character used here for superquoting is the apostrophe. E.g.
PRINT(QUOTE '()) will print a single opening parenthesis.
PRINT QUOTE ')) will print a space
PRINT(QUOTE '')) will print an apostrophe
Even a digit sequence will not be considered a number when the first digit is superquoted, because this now becomes an ordinary character of an identifier. (PRINT(QUOTE('12345678987654321)) will print '12345678987654321 although the number capacity for arithmetic numbers is perhaps greatly exceeded. The arithmetic functions do not calculate with such digit sequences!

Input is done with the function READ having no arguments. Because of the conventions for writing functions, a call of READ in inner lisp still has to be written as (READ) otherwise it would be regarded as a variable with no value. The value of (READ) is a single list (special case: a single atom) read from the input device. READ always reads a balanced string.

But READ performs an extra service for us. Every atom read has to be checked for uniqueness as we agreed that an atom with a particular identifier has to be unique. For this purpose a list is kept which is called the OBLIST. Every atom read by READ is looked up in the OBLIST. If its identifier is not yet there, it is added to the list and a new cell is created by CONS for that atom (together with all the space needed for the identifier). But if the identifier is already in the OBLIST, the pointer to that atom is returned. The only chance of creating a different atom with the same name is to remove the first one from the OBLIST. This does not mean that the atom will be destroyed. It could very well be part of some other property list.

A useful function belonging in the class of input-output functions is TERPRI, no parameters. Its name is derived from Terminate Printing. It emptied the buffer of line printer onto that printer and set up a new line. Here the output is immediate on PRINT,and TERPRI only gives the new line. One could define:

```
EX(TERPRI NIL (PRINT (QUOTE '
)))
```

The new line character is invisible in itself but has to be superquoted.

1.14 The structure of the OBLIST

The OBLIST is itself an object on the OBLIST. In the beginning the atom OBLIST has the following property list:

(APVAL(NIL OBLIST))

After having read e.g. the atoms A and B the OBLIST now looks:

(APVAL(NIL B A OBLIST))

The NIL is present for technical reasons which will become clear in the chapter on RPLACA and RPLACD.

As OBLIST is itself an object in its own OBLIST it is essentially a circular object. As the circularity goes through the atom header of OBLIST this is never apparent. CDR(OBLIST) will give (APVAL(NIL ... OBLIST)) and EVAL(OBLIST NIL) will give (NIL ... OBLIST) as expected.

1.15 EVALQUOTE

We are now in a position to explain for the first time the action of the system as it presents itself to us on the keyboard. The system is expressible as a function which is usually called EVALQUOTE.

```
EX(EVALQUOTE() (CONS (PRINT(APPLY(READ)(READ)NIL))
                     (EVALQUOTE) ))
```

EVALQUOTE is a function of no parameters. If it is started up for the first time with EVALQUOTE NIL the action is:
It will read twice. The item read first will be applied to the arguments read as the second item with an initial association list NIL, the result of this wil be printed and the second argument of CONS will recursively call EVALQUOTE again. It is clear from this description that CONS will never come to action. We could have taken any other function of two arguments. The two arguments were only used to do two actions in succession. There is a big disadvantage in this system. Of course it will keep the return to the outside recursion forever and within a short time the whole store will be used up for a stack of returns which will never be used.

In the next chapters we shall see how we can overcome this drawback.

The use of APPLY in the above definition of EVALQUOTE implies that the arguments given are taken as quoted. They are not evaluated. We also could have written another EVALQUOTE for inner LISP as follows:

```
EX(INNERLISP() (CONS(PRINT(EVAL(READ)NIL)) (INNERLISP) ))
```

Or better still we could build a system which will distinguish inner and outer lisp itself by defining:

```
EX(EXECUTE(X)(COND((ATOM X)(PRINT(APPLY X (READ) NIL)))
                  ((EQ(CAR X)LAMBDA)(PRINT(APPLY X (READ) NIL)))
                  (T(PRINT(EVAL X NIL))) ))
EX(INNEROUTER()(CONS(EXECUTE (READ))(INNEROUTER)))
INNEROUTER NIL
```

The system will now both recognize
CAR((A B)) which gives A and also
(CAR(QUOTE(A B))) also giving A. Even if the first object typed is not atomic but a lambda form, it will be recognized as such.

1.16 The PROG feature

As we have seen, many things can be done recursively but from the standpoint of the machine it is not always the most efficient way. Doing processes iteratively will take no space for storing the returns from all levels of recursion for example. Perhaps a more important point from a pragmatic reason is that programmers are more used to the method of sequential programming with go to and that that is the main reason why PROGs have come into LISP.

A sequential program can be expressed in LISP by
(PROG (list of prog variables) labels and (statements) in an arbitrary number and sequence)
Labels in this sense are plain atoms, statements are function calls hence sublists. The statements are executed one by one in the list sequence and their value is lost. The program variables are almost treated in the same way as formal parameters in a lambda-form. The difference is that they do no correspond to an actual parameter and they are initially associated with NIL. They are placed on the association list just as formals.

Without a few more functions one could do very little with PROGs. The new functions needed are
(GO label) will go to the mentioned atomic label. This label must be present within the PROG, otherwise an error message will result. The label may not lie outside of this PROG. Jumping is strictly local within a single PROG level. Hence one cannot jump inside an inner nested PROG and one cannot jump outside to a surrounding level. This makes PROG still manageable as a function when looked upon from the outside. The difficult scope problems are not present. E.g. in ALGOL jumping out of a block necessitates the reinstallation of the environment as it was at the place to where the jump was made.
(RETURN form) To finish the execution of a PROG a RETURN is made. It will return with the value of the form as the value of the whole PROG. A RETURN of a GO outside a PROG will be signalled as an error.

A COND inside a PROG has also a different behaviour. While in a function a COND was not allowed to fall through the bottom, in a PROG this is allowed. The PROG simply goes on with the next statement. Furthermore we shall extensively use the following simplification. The object T is defined DEFLIS(((T T))APVAL) EX(T(X)X) that is: the object T has the property list (APVAL T EXPR(LAMBDA(X)X)) . Hence, T applied as a function to an argument will deliver the value of the argument as its value. And NIL applied to an argument produces NIL and does nothing.

A condition in a PROG can now be simply written as a pair. If the first element is T, the second element is executed and if the first element is NIL the result is NIL and nothing is done.

Before a good use can be made of PROG we certainly need a form of assignment. As the result of a function is lost when executed as a statement in PROG, we need a mechanism to assign this value to a variable. This will be treated in the next chapter.

In any case we are now in a better position to write another version of EVALQUOTE.

```
EX(EVALQUOTE NIL (PROG NIL A (EXECUTE(READ)) (GO A)))
```

This function will do the read, execute, print cycle in an iterative way for an indefinite number of times and without building up a useless stack of returns.

1.17.Functions with an indeterminate number of arguments

LISP has a very specific method to treat and define functions with an indeterminate number of arguments. We have met already the functions PLUS and LIST but we could not define them yet. If we want to define a function with an arbitrary number of arguments we do this in the following way:

DEFLIS(((functionidentifier(LAMBDA(L A)(body))...)FEXPR)

The indicator now is FEXPR, the lambda form will always have two formal parameters, the first of which will be the list of all the unevaluated actual parameters and the second will be the association list in force at that moment. As actual parameters in a call an arbitrary number of arguments will be given. The system will pack these arguments without doing any evaluation and associate them to the first formal parameter, the present association list will be automatically supplied as the second parameter. The user even could not do that as the association list is not normally accessible to him but is kept updated by the system.

We now have gained three possibilities in one stroke:

a) We can have functions with an arbitrary number of arguments
b) The arguments are not evaluated but so to say automatically quoted.
c) We can get hold of the association list at the moment. This will be needed for evaluating some or all unevaluated arguments within the body.

To make it a little bit easier for ourselves we shall introduce the function FEX to define this kind of functions with the indicator FEXPR. As the formal parameter list always consists of two parameters we shall always take (L A) where L stands for the list of unevaluated parameters and A for the association list.

EX(FEX(NAME BODY) (DEFLIS(LIST(LIST NAME(LIST LAMBDA (QUOTE (L A)) BODY)))FEXPR))

This will do the 3 parenthesis packing, the insertion of LAMBDA and (L A) and the supplying of FEXPR for us.

Examples:

The function QUOTE can now be defined as

FEX(QUOTE(CAR L)) i.e. QUOTE will deliver the first unevaluated argument from the list of arguments. Note that there is no test on the correct number of arguments now; it simply disregards all arguments beyond the first. If we suppose that there is a universal function ERROR to call when an error is made, we could write:

```
FEX(QUOTE(PROG NIL ((NULL L)(ERROR))
                   ((NOT(NULL(CDR L)))(ERROR))
                   (RETURN (CAR L)) ))
```

Another example of FEXPR's is

```
FEX(ALIST A)
```

As the formal parameter is the present association list we now have created a function to be called as (ALIST) to look into it.

Let us see now how we can use FEXPR's for defining functions of an arbitrary number of arguments e.g. LIST.

One would be inclined to write:

```
FEX(LIST(COND((NULL L)NIL)
         (T(CONS(EVAL(CAR L)A)(LIST(CDR L)) )) ))
```

But this is wrong! If there are no arguments the result is NIL, otherwise CONS the explicitly evaluated first argument (a FEXPR has not done the evaluation for us!) to the list of the remainer. But alas, as LIST is a FEXPR the argument (CDR L) is not evaluated and is taken literally.

The error is prevented in the following way:

```
FEX(LIST(EVLIS L A))
EX(EVLIS(L A)(COND((NULL L)NIL)
         (T(CONS(EVAL(CAR L)A)(EVLIS(CDR L)A))) ))
```

Now LIST calls in the use of an ordinary EXPR and simply transmits the list of unevaluated parameters and the association list to that other function.

Suppose we had only a function PLUS2 which could add two numbers (as did SUM in chapter 1.11). Now PLUS for an arbitrary number of arguments can be defined by

```
FEX(PLUS(EVPLUS L A))
EX(EVPLUS(L A)(COND((NULL L)NIL)
  ((NULL(CDR L))(EVAL(CAR L)A))
  (T(PLUS2(EVAL(CAR L)A)(EVPLUS(CDR L)A))) ))
```

We are also able to define a function IF which more resembles the ALGOL <u>if</u> . We shall first do it the wrong way.

```
EX(IF(A THEN B ELSE C)(COND A B)(T C)) )
```

Suppose we apply this to define LENGTH

```
EX(LENGTH(L)(IF (NULL L) THEN 0 ELSE (PLUS 1 (LENGTH(CDR L))) ))
```

There are two errors now. If we try to use this function it will give the error message: THEN has no value. The same hold for ELSE. This can be circumvented by defining DEFLIS(((THEN THEN)(ELSE ELSE))APVAL). Now they have a value which is themselves. As they are not really used inside IF this does not matter.

The second error is by far the more serious one. A function will evaluate all its arguments beforehand. So does the first recursion etc. and this will never end.

The solution is a FEXPR, which does not evaluate its arguments

```
FEX(IF(COND((EVAL(CAR L)A)(EVAL(CADDR L)A))
  (T(EVAL(CADDDDR L)A)) ))
```

Here the 3rd or the 5th argument is evaluated explicitly and only the required one. Also THEN and ELSE are not evaluated.

1.18. Assignment

At last we have the complete arsenal ready for defining assignment. At function call a value was bound on the formal by adding the pair on the association list. Now we want to be able to change these bound values. To this end we shall first define a function to search a name on the association list.

```
EX(ASSOC(X A)(COND((NULL A)NIL)
  ((EQ(CAAR A)X)(CAR A))
  (T(ASSOC X(CDR A))) ))
```

If the association list is empty, we return with NIL. If the name part of the first pair is the name looked for we return with the complete pair. The value can easily be extracted by CDR.

Now comes the essential part of the assignment. We must be able to break the link between the name and the value associated and replace it by another value. Until so far we have only been able to inspect existing lists and to build new ones with CONS. We could of course reconstruct the association list with a new value but this would be extremely costly in time. Therefore there are two functions which actually break a link by inserting another one.

(RPLACA X Y) will replace the CAR link of X by Y and
(RPLACD X Y) will replace the CDR link of X by A. For example:
RPLACE((A B) C) will give (C B)
RPLACD((A B) C) will give (A.C)
RPLACD((A B) (C)) will give (A C)
It is as if we had (CAR X) := Y and (CDR X) := Y .
We shall call these functions the dangerous functions. They have to be handled with great care. There are some variants of LISP which are having this mechanism as their primary feature. E.g. WISP, a compiled version of a list processor devised by Wilkes is based on this. [44]

Two kinds of assignment for general use will now be defined.

```
FEX(SET(RPLACD(ASSOC(EVAL(CAR L)A)A)(EVAL(CADR L)A))))   and
FEX(SETQ(RPLACD(ASSOC(CAR L)A)(EVAL(CADR L)A))))
```

The function SET evaluates on both sides. This is quite unusual for ALGOLians. Suppose that X had the value Z and Y had then value 3. Now (SET X Y) would have the effect that Z has become 3. This function is seldom used.

Note how useful it was to have ASSOC come back with the complete pair of name and value, so that RPLACD could change its value field. Because ASSOC needed the association list, it was handier to define SET straight away as a FEXPR. It could have been defined as an EXPR by

```
EX(SET(X Y)(RPLACD(ASSOC X (ALIST))Y) )
```

The two arguments are now evaluated automatically and the previously defined function ALIST supplies the association list.

The function SETQ will not assign to the value of the first argument but to the name itself. This corresponds completely with e.g. ALGOL 68 (and many other languages), where the left hand part of an assignment is only a name, i.e. a reference to a value and the right hand part is a value which supersedes the value to which the left hand part referred originally.
One can regard SETQ as a function implicitly taking the left parameter QUOTEd. E.g. one can write (SET (QUOTE X) 3) to assign 3 to X itself. Hence its name SETQ. One could be tempted to write

```
EX(SETQ (X Y) (SET (QUOTE X) Y))
```

but this would be totally wrong. Not the formal parameter X should be quoted but the actual parameter!

1.19 Examples of PROG with SETQ

A great many examples now become a lot easier to express and often more efficient. Without too much comment we shall give now redefinitions of several older functions we have met before in the PROG style.

```
EX(LENGTH(X)(PROG(K)
  (SETQ K 0)                          The prog variable K was NIL initially
A ((NULL X)(RETURN K))            A:  if null(X) then return(K)
  (SETQ K(PLUS K 1))                  K := K + 1
  (SETQ X(CDR X))                     X := rest(X)
  (GO A) ))                           go to A
```

Especially REVERSE can be written much more perspicuous with PROG.

```
EX(REVERSE(X) (PROG(Y)                Y is automatically initialized to NIL
A ((NULL X)(RETURN Y))
  (SETQ Y(CONS(CAR X)Y))              Unstack X, stack on Y
  (SETQ X(CDR X))
  (GO A) ))
```

The function PRINT can now be defined in terms of a more primitive function PRINTATOM.

```
EX(PRINT(X)(PROG(K)
      ((ATOM X)(GO ATOM))                Any name may serve as label
      (SETQ K X)                         Make working copy
      (PRINTATOM(QUOTE '( ))             Print (
LIST  (PRINT(CAR K))                     Recursive use of PRINT
      ((NULL(CDR K))(GO CLOSE))
      ((ATOM(CDR K))(GO DOT))            Dot notation for atomic end
      (PRINTATOM(QUOTE ' ))              Space
      (SETQ K(CDR K))                    Next in list
      (GO LIST)
DOT   (PRINTATOM(QUOTE '.))              Print .
      (PRINT(CDR K))
CLOSE (PRINTATOM(QUOTE ') ))             Print )
      (RETURN X)                         Return with original value
ATOM  (PRINTATOM X)
      (RETURN X) ))
```

We see here a very efficient mixture of recursion and iteration. For the CAR direction recursion is used as this is the most natural way of expressing descent of a level, for progressing in the list direction iteration is used. In general, lists are longer in the CDR direction than in the CAR direction.

We can now also redo the bad example of the Fibonacci series in chapter 1.10. We shall make a function FIB2 which will return a dotted pair consisting of the required value and the previous value, produced as a by-product.

```
EX(FIB(X)(CAR(FIB2(X)))                    The other value is not needed
EX(FIB2(X)(PROG(K)                         We need K to hold something
  ((EQ X 0)(RETURN(CONS 0 0))
  ((EQ X 1)(RETURN(CONS 1 0))              The exceptions first
  (SETQ K (FIB2(MINUS X 1)))               Will produce  (fib(x-1).fib(x-2))
  (RETURN(CONS(PLUS(CAR K)(CDR K))(CAR K)) )) Return with fib(x-1)+fib(x-2) and
                                             the previous value
```

This process is at least linearly efficient. Of course still better the direct formula for the n'th term or a logarithmically efficient process could have been written. This is left as an exercise to the reader.

1.20. Editing functions

In any conversational language typing errors should be easily corrigible.. We shall give a set of functions called CHANGE, DELETE, INSERT and CHANGEALL with the following action

CHANGE(A B C) will change the first appearance of A (literally) in the property list of B into C. It will return with T when succeeded

DELETE(A B) will delete the first appearance of A in the property list of B

INSERT(A B C) will insert after the first appearance of A in the property list of B the insertion C

CHANGEALL(A B C) will change all appearances of A in B by C.

The functions give again an application of the technique of combined recursion in the CAR direction and iteration in the CDR direction.

First an auxiliary function will be defined to search for an appearance of A in the property list of B.

```
EX(SEARCH(A B)(PROG(K)
  (SETQ W NIL)                              A global switch to indicate the
A ((ATOM B)(RETURN NIL))                    beginning of a list
  ((EQUAL A(CAR B))(RETURN B))              Return with the whole list, not only
  (SETQ V B)                                the object. V is another global
  (COND((SETQ K(SEARCH A(CAR B)))(RETURN K)))   An intermediate assignment
  (SETQ B(CDR B))                           the result of which is tested. As
  (SETQ W T)                                it is not always T, COND is necessary
  (GO A) ))                                 W is set true once in the list
                                            direction.

EX(CHANGE A B C)(PROG (K V W)               Here, K V and W are introduced
  ((NULL(SETQ K(SEARCH A(CDR B))))          If not found return NIL
    (RETURN NIL))
  (RPLACA K C)                              Replace first appearance
  (RETURN T) ))                             and return with T

EX(DELETE(A B)(PROG(V W)
  ((NULL(SEARCH A(CDR B)))(RETURN NIL))     Return NIL if not found
  (W(GO B))                                 Test switch set by SEARCH
  (RPLACA V(CDAR V))(RETURN T)              If first of list, use RPLACA
  (RPLACD V(CDDR V))(RETURN T) ))           If not first, use RPLACD
```

The first in a list comes from a next higher level with CAR, the next in a list comes from the previous place with CDR

```
EX(INSERT(A B C)(PROG K V W)
  ((NULL(SETQ K(SEARCH A(CDR B))))(RETURN NIL))
  (RPLACD K(RPLACD(LIST C)(CDR K)))(RETURN T) ))
```

The new element is joined into the list with two RPLACD's

The shortest function of the set is:

```
EX(CHANGEALL(A B C)
  ((CHANGE A B C)(CHANGEALL A B C)) )
```

First,the first appearance is changed and if this succeeds CHANGALL is called upon to try again. The ultimate result is NIL.

After having defined these functions

```
DELETE(SEARCH OBLIST)
```

will delete the atom SEARCH from the OBLIST. The function itself is not destroyed as it still hangs on the property list of CHANGE. But somebody else might want do define his own version of SEARCH for quite another purpose. In this way an identifier may be protected. Nevertheless printing out the property list of CHANGE will show SEARCH. The atoms A, B, C, K, V, W have no properties and cannot do any harm. The space will not be saved; on the contrary, if they would be removed from the OBLIST, another user would have to create a new atom A etc.

1.21. Tracing

Although programming should be done in such a systematic way that the method of making the program is almost a proof in itself, we are faced in practice with thinking errors which are difficult to trace without computer assistance. A simple way to do this tracing is to redefine the functions to be traced together with some PRINT statements. Suppose we want all CAR's traced and print their argument and result. This could be done by

```
EX(CAR(X)(PROG NIL
  (TERPRI)(PRINT(QUOTE CAR))              Name of the function is printed
  (PRINT X)                               Print argument
  (PRINT(QUOTE =))
  (RETURN(PRINT(CAR X))) ))               Print result.
```

This method is hardly to be called automatic. We shall present here a set of functions TRACE and UNTRACE which can take an arbitrary number of arguments, all names of functions to be traced or restored to their original form. In many LISP implementations this is built in. Here we shall do it by rebuilding a new EXPOR or FEXPR property and storing the original property under another indicator. To gain access to the property list use is made of (GET name indicator) which will get the property under the given indicator from the given atom name. It is a built-in function but could be defined by

```
EX(GET(OBJ IND)(PROG(K)
  (SETQ K(CDR OBJ))                       Take whole property list of OBJ
A ((NULL K)(RETURN NIL))
  ((EQ(CAR K)IND)(RETURN(CADR X)))        If found return with property
  (SETQ K(CDR K))  (GO A) ))
```

We shall now give the functions TRACE and UNTRACE and leave the study of it to the reader.

```
FEX(TRACE(PROG X Y Z N)                   N is used for recursion depth
  (DEFLIS(LIST(LIST(QUOTE N)0))APVAL)     N := 0
A ((NULL L)(RETURN(QUOTE READY)))
  (SETQ X(CAR L))
  (COND((GET X(QUOTE *))(GO B))           * is used as special indicator
    ((GET X FEXPR)(GO C))                 Do not trace traced functions again
    ((GET X EXPR)(GO C))                  Only EXPR's and FEXPR's are traced
  (GO B)
```

```
C (SETQ Z(CADADDR X))                          Take body
  (RPLACD(CDR X)(CONS(QUOTE *)(CDDR X)))
  (SETQ Y(LIST LAMBDA Z(LIST PROC(LIST(QOUTE **))
    (LIST(QUOTE P)(QUOTE N)1 X)
    (LIST SETQ(QUOTE **)(CONS(LIST CET(LIST QUOTE X)
       (LIST QUOTE(QUOTE *)) )Z))
    (LIST(QUOTE P)(QUOTE N) -1 X)              Rebuild body
    (LIST RETURN(LIST PRINT(QUOTE **))) )))
  (RPLACD(CDR X)(CONS Y(CDDR X)))
B (SETQ L(CDR L))  (GO A) ))

FEX(UNTRACE(PROG K M)
A ((NULL L)(RETURN NIL))
  (SETQ K(CAR L))
  (COND((SETQ M(GET K(QUOTE *)))(RPLACD(CDR K)(LIST M))))
  (SETQ L(CDR L))                              Replace property list by previous one
  (GO A) ))
```

An auxiliary function P is called upon. This function contains (LESSP X Y) which has the same meaning as X < Y in ALGOL.

```
FEX(P(PROG(K)
  (DEFLIS(LIST(LIST(CAR L)(PLUS(GET(CAR L)APVAL)(CADR L))))APVAL)
  (TERPRI)                                     Increase or decrease recursion
  (SETQ K(EVAL(CAR L)A))                       depth counter
A ((LESSP K 1)(GO B))                          Do function
  (PRINT(QUOTE -))                             Print as many - signs as recursion
  (SETQ K(MINUS K 1))                          depth to show indentation
  (GO A)
B ((LESSP(CADR L)0)(PRINT(QUOTE =)))           = printed for result
  (PRINT(LIST(CADDR L))) ))
```

At the end we give

```
EVAL((RPLACD OBLIST(QUOTE(TRACE UNTRACE OBLIST)))NIL)
```

which will rebuild the OBLIST and will only keep TRACE and UNTRACE. All other identifiers are made inaccessible, they are "protected" in terms of ALGOL 68.

TRACE and UNTRACE both take an arbitrary number of arguments being the names of the functions to be traced. It will be clear that these functions must be ordinary EXPR's or FEXPR's. No built-in function can be traced that way. There is nevertheless a method to extend the tracing to standard functions such as CAR, CDR etc. Before entering the function in which the standard function to be traced are appearing, the standard functions are redefined first. E.g. if CAR has to be traced then first

```
EX(CAR(X) (CAR X))
```

is given. This has redefined CAR as an EXPR in terms of the standard function which is now inaccessible under that same name. Now any function using CAR will find the redefined CAR and this CAR can be traced by

```
TRACE(CAR)
```

One has to be very careful in tracing FEXPR's. As the evaluation is done at another moment, it could happen in a wrong environment. We shall see examples of this in the chapter on functional arguments, where the protection problems will be treated.

For examples see appendix 2.

1.22 Functional arguments

Until so far we have only used arguments, which were only values not functions. We can define the following function

```
EX(MAPCAR(FN L) (COND
  ((NULL L)NIL)
  (T(CONS(FN(CAR L))(MAPCAR FN(CDR L)))) ))
```

This function is very useful to apply a function, that was meant for one argument only,to a list of arguments and produce a list of results.
Suppose we have the function

```
EX(INCREMENT (X) (PLUS X 1) )
```

and we want to use this function to increment all elements of a list. We could write:

```
(MAPCAR (QUOTE INCREMENT) (QUOTE (1 2 5 10))) resulting in (2 3 6 11)
```

We have to quote, otherwise the system would say: INCREMENT has no value.
We could as well have written the lambda form directly

```
(MAPCAR(QUOTE (LAMBDA (X) (PLUS X 1))) (QUOTE (1 2 5 10)) )
```

Nevertheless there is something wrong in the general case as will be shown in the following example
Suppose we execute

```
EVAL((G P Q (QUOTE (LAMBDA(S)(CONS S Y))))  ((P.IS)(Q.WRONG)(Y.RIGHT)))
```

Where G has been defined as

```
EX(G(X Y FN) (PROG NIL
  ((NULL X)(RETURN NIL))
  ((NULL Y)(RETURN NIL))
  (PRINT(ALIST))  }                This statement could be inserted for
  (RETURN(FN X) ))                 inspection of the association list
```

We would expect at first sight that the result would be (IS.RIGHT). But we will see (IS.WRONG) coming out! Let us inspect the association list. It will look

```
((FN LAMBDA(S)(CONS S Y))(Y.WRONG)(X.IS)(P.IS)(Q.WRONG)(Y.RIGHT))
```

When Y is looked up we expected to find the bottom association (Y.RIGHT) but of course (Y.WRONG) is met earlier by ASSOC and the wrong binding will show up. It was caused by using Y twice, in the definition of G and in the EVAL.

There is a mechanism in LISP to preserve the environment as it as at the time of definition. It is called the FUNARG mechanism and will be explained in detail when the inside working of the system is dealt with. Let it suffice now to give the remedy. Functional arguments must be quoted not by QUOTE but by FUNCTION (we shall use FUNCTI in this implementation).

Let is do new again the previous example but with FUNCTI

```
EVAL((G P Q(FUNCTI(LAMBDA(S)(CONS S Y)))) ((P.IS)(Q.WRONG)(Y.RIGHT)))
```

We now get the correct result (IS.RIGHT)
The association list as printed by (PRINT(ALIST)) would have looked:

```
((FN FUNARGS((S)(CONS S Y))(P.IS)(Q.WRONG)(Y.RIGHT))(Y.WRONG)(X.IS)(P.IS)
 (Q.WRONG)(Y.RIGHT))
```

We will see that the function FN is bound to the lambda form (without LAMBDA) under a special indicator FUNARG together with the association list as it was at the time of the definition. That association list is reinstalled when FN is put to action. The casual user will never see the atom FUNARG as it is automatically handled by FUNCTI. It would seem that the association list has become much longer now, but we have to remember that the association list at the time of definition under the indicator FUNARG is shared with the tail of the association list. It will only cost a reference!

Similar problems exist in many languages. E.g. in ALGOL we could write

```
begin real x; proc y = real : x ; | x is procedured to mode proc real |
     begin real x; | a local x, totally different from the x outside |
                   ... ; x := y ; ...   | here the local x is assigned the
```

value of the non-local x, with the help of procedure y. This procedure y must insert a correct clause with the proper environment.

It is sometimes said that a good LISP programmer can be recognized from the frequency of using MAPCAR and functional arguments. Here is another example giving extremely compact programming for a Cartesian product.

```
EX(INDEX(L END FN)(COND
  ((NULL L)END)
  (T(FN(CAR L)(INDEX(CDR L)END FN))) ))
```

An auxiliary function to step through a list.

```
EX(CARTESIAN(S1 S2)
 (INDEX S1 NIL(FUNCTI(LAMBDA(I1 J1)
     (INDEX S2 J1(FUNCTI(LAMBDA(I2 J2)
       (CONS(CONS I1 I2)J2) ))) ))) ))
```

If we give CARTESIAN((A B C D)(1 2 3 4 5)) this will result in

```
((A.1)(A.2)(A.3)(A.4)(A.5)(B.1)(B.2)(B.3)(B.4)(B.5)
 (C.1)(C.2)(C.3)(C.4)(C.5)(D.1)(D.2)(D.3)(D.4)(D.5))
```

A famous problem in the literature of ALGOL[37] is the problem by Knuth, Man or Boy. It was used to test whether one had a Man compiler or only a Boy compiler. Not only the problem of recursion and cross-recursion appeared in it but also the problem of the recursive non-locals. It is comparatively easy to create recursive locals on a stack implementation. But we shall see that also non-locals (which could be formal or local in a bigger block) can go into recursion. Te problem can be stated in ALGOL 60 as follows:

```
begin integer procedure A(K, X1, X2, X3, X4, X5); value K; integer K;
     begin integer procedure B;
          begin K := K - 1 ; A := B := A(K, B, X1, X2, X3, X4)end;
          if K <  0 then A := X4 + X5 else  B
     end;
     print(A(6,1,2,3,4,5))
end
```

An ALGOL purist could object to executing B as a procedure and not as a function designator. In that case we could declare int M and write M := B to satisfy him. Remark also that the name parameters are not specified. This is not essential. A parameterless procedure could go in as well as an integer expression. The point of the procedure A is that (when K > 0) it will call B and via B again A with parameters, which are shifted down over one place, X1 being replaced by B. Now it is very important to remember the environment of B as B makes use of non-local quantities, changing with the environment. For high enough K we will ultimately execute A := B + B. But which B's are meant?

Of course this problem can be solved very elegantly in LISP and the FUNARG mechanism will preserve the proper environment.

```
EX(A(K X1 X2 X3 X4 X5)(COND
  ((LESSP 0 K)(B))
  (T(PLUS(COND((NUMBER X4)X4) (T(X4)))
         (COND((NUMBER X5)X5) (T(X5))) )) ))
EX(B() (A(SETQ K(MINUS K 1))(FUNCTI B)X1 X2 X3 X4))
```

LISP does not have to declare the function B within A. LISP is also not very fussy about using a function as a procedure. But it must make a difference between a number, which is evaluated as X and a parameterless function which is evaluated as (X). Also note the intermediate assignment to K. Now A(6 1 2 3 4 5) will nicely evaluate to 28 as should be.

See Appendix 2 for a trace.

Closely related to the problem of preserving the proper environment is the problem of partial parametrization. It could be very useful to supply a function with only some of the required parameters. The result should be a function with less parameters, but still a function. In principle this can be done in LISP but the problems are formidable. Because the actual text of the definition is available, the text could be rebuilt into a new lambda form. This has been done in the past [40, 41, 42].

ALGOL 68 has not succeeded in finding a proper wording to have this in the language just because of scope and environment problems. But POP-2, an ALGOL like, conversational language, has a very nice solution. They have disregarded the ALGOL tradition to preserve the old environment, but they take the most recent incarnation of a value on the association list as we would get in LISP when quoting a functional argument by QUOTE instead of FUNCTI. E.g. we would like to define

fourthroot := twice(sqrt)

with a function twice. This function twice can be defined in POP-2.

We shall follow the POP notation. A function definition is given in the form function fn parameters; body end
and a lambda form is lambda parameters; body end
If we would write now

function twice f; lambda x; f(f(x)) end end;

this would not work. The value of f is only locally bound to twice and will only be available during the execution of twice. Hence they use the technique of partial parametrization to "freeze in" the value of f. To indicate such a partial list of parameters, they use special brackets, so-called decorated brackets, written as (% and %) . The parameters will always be right adjusted leaving the parameters on the left as outside parameters. We not define

function twice f; lambda x f; f(f(x)) end (% f %) end;

and this now works. If we now have

function add1 x; x + 1 end; then we can write
add2 := twice(add1); and add2(5) would give 7

twice(add1)(5) would be wrong but (twice(add1))(5) would be right. There is even a function to get at the frozen values by giving frozval(n,other) which changes the n'th frozen value by the value other.

Alos a functional product can be defined as

function funprod f g; lambda x f1 g1; g1(f1(x)) end (% f, g %) end;
nonop ** := funprod; This has assigned the procedure funprod to an operator ** and now we can write (sin ** cos)(argument).

1.23. Changing the notation

LISP has a rather ugly notation. We will show how the notation can be changed very quickly by some examples.

Let us first redefine IF so that it can take an arbitrary number of following IF clauses. E.g. (IF A THEN B ELSE C) but also (IF A THEN B ELSE IF C THEN D ELSE E) must be taken. We shall not insist on testing the syntactic presence of the words THEN and ELSE in their correct positions. This is left to the reader as an exercise.

```
FEX(IF(PROG NIL
A ((EVAL(CAR L)A)(RETURN(EVAL(CADDR L)A)))        If first arg is true, take
                                                  third argument
  ((EQ(CADDDDR L)(QUOTE IF))(GO B))               If fifth arg. is IF, repeat
  (RETURN(EVAL(CADDDDDR L)A))                     Else do ELSE clause
B (SETQ L(CDDDDDDR L)) (GO A) ))                  Cut off X then Y ELSE IF
```

Next we need a translator, which can recognize operators in the usual infix notation an translate them to prefix notation as is used in LISP. We shall assume that operators have the indicator OPERATOR on their property list together with the name of the corresponding function name for that operation.

```
DEFLIS(((* TIMES)(+ PLUS)(- MINUS)(< LESSP)(= EQUAL)(:= SETQ))OPERATOR)
```

The following translator TR will do the conversions from X * Y to (TIMES X Y). There will be no priorities for the operators yet for reasons of demonstrability and simplicity.

```
EX(TR L) (PROG(K)
  ((ATOM L)(RETURN L))                    An atom is untranslated
  ((NULL(CDR L))(GO B))                   If tail is empty, then no triple
  (SETQ K(GET(CADR L)(QUOTE OPERATOR)))   Look up whether operator and get it
  (COND(K(GO A)))                         If so, goto A
B (RETURN(CONS(TR(CAR L))(TR(CDR L))))    If not, treat sublists recursively
A (RETURN(CONS K(CONS(TR(CAR L))(TR(CDDR L))))) ))
                                          Compose a new list, consisting of
```

translated operator, first object and third object. Both objects are submitted to another pass through TR as they could contain further operators.

To be able to apply this TR easily we further define

```
EX(TRANS(NAME FORMALS BODY)
 (DEFINE(LIST(LIST NAME(LIST LAMBDA FORMALS (TR BODY))))) )
```

Here follows an example of simple definitions of TIMES.

```
TRANS( TIMES(X Y)
  (IF (X = 0) THEN 0 ELSE IF (Y = 0) THEN 0 ELSE (Y + ((X - 1) * Y)) ) )
```

Although still a lot of parenthesis are needed, this looks much better than before.

Note that the operator * is used in the translation even before the function TIMES has been defined. The translation here is only a partial one, as IF is not translated but interpreted as a function. The translation would look as follows:

```
(EXPR(LAMBDA(X Y)(IF(EQUAL X 0)THEN 0 ELSE IF(EQUAL Y 0)THEN 0
  ELSE(PLUS Y(TIMES(MINUS X 1)Y))) ))
```

Another faster definition would be

```
TRANS(TIMES(X Y)  (PROG(Z)
  (Z := 0)
  ((Y < X)(RETURN (Y * X))))
A : ((X = 0)(RETURN Z))
  (X := (X - 1))
  (Z := (Z + Y))
  (GO A)  ))
```

This looks almost like ALGOL. In the same style we write

```
TRANS(FAC(X)  (IF(X = 0)THEN 1 ELSE (X * (FAC(X - 1))) ))
```

In the next chapter we shall use macros to make a more complete compilation.

1.24 Macros

The way in which IF was defined in the previous chapter did not really change the text of the program. What we really want is change (IF A THEN B ELSE C) into another text (COND(A B)(T C)) . This is the principle of compilation. In fact we compiled the infix notation (X * Y) into (TIMES X Y). Of course it is immaterial whether we compile into LISP or into another language, e.g. machine language. Then (X * Y) would be compiled perhaps to

```
LDA X
MUL Y
```

in some obvious but purely fictitious machine code.

If some definite construct in the source language will give rise to another definite construct in the target language we can speak of a macro. A macro is a shorthand writing for a target construct and the transformation is done before the program is run, i.e. the macro expansion takes place at compile time. Now this is a dangerous distinction. In a language such as LISP, where internal objects have the same form as external objects the system can change over from compilation to running at any moment. But let us assume that we shall keep these phases separate for the moment.

There is some superficial similarity between functions and macros. If we write (IF A THEN B ELSE C) we want to compile to (COND(A B)(T C)) but if we write (IF P THEN Q ELSE R) we want to compile to (COND(P Q)(T R)). In other words, there are also formal parameters in the description, which have to be replaced by actual parameters, although they are now replaced in a strict textual sense.

We want to define this kind of macro by

MACRO(IF(X Y Z) (COND(X Y)(T Z))) in the same style as with EX but with a totally different action. We shall call this kind of macros prefix macros as the name of the macro is coming first.

Another kind of macro is the infix macro, generally a macro for defining an operator. Here the source construct has the name of the macro between two parameters. We shall define them by e.g.

INFIX(+ (X Y) (PLUS X Y))

Both kind of macros could be termed "model macros" as in the definition a complete model is given of the target construct. Only the parameters have to be substituted in the text.

A completely different kind of macro we shall term "recipe macro". Here the target construct is completely dependent on the parameters and the result has to be concocted from the ingredients by a recipe.

This kind of macro could be defined by
RECIPE(F(parameters) (body of recipe))

We shall first make a simple version of a universal macro expander which we shall call ME. For model macros this macro expander has to call upon a copying routine which will copy the model but replace all formal by the corresponding actual parameters.

```
EX(COPY(BODY FORMALS ACTUALS) (PROG NIL
  ((ATOM BODY)(GO A))                        If atomic it could be a formal par.
  (RETURN(CONS(COPY(CAR BODY)FORMALS ACTUALS)      If not, break it down
              (COPY(CDR BODY)FORMALS ACTUALS)))    recursively and reconstruct
A ((NULL FORMALS)(RETURN BODY))              Return if formals exhausted
  ((EQ BODY(CAR FORMALS))                    If a formal is spotted, replace it
   (RETURN(CAR ACTUALS)))                    by corresponding actual
  (SETQ FORMALS(CDR FORMALS))
  (SETQ ACTUALS(CDR ACTUALS))
  (GO A) ))
```

It is clear from this definition that too many actuals are disregarded, while too many formals will result in the excess formals being replaced by NIL. The three kinds of macros will be defined by attaching the indicator MACRO, INFIX and RECIPE resp.

```
EX(MACRO(X Y Z) (DEFLIS(LIST(LIST X(LIST LAMBDA Y(ME Z))))(QUOTE MACRO)) )
EX(INFIX(X Y Z) (DEFLIS(LIST(LIST X(LIST LAMBDA Y(ME Z))))(QUOTE INFIX)) )
EX(RECIPE(X Y Z)(DEFLIS(LIST(LIST X(LIST LAMBDA Y(ME Z))))(QUOTE RECIPE)))
```

The main function is ME, the macro expander.

```
EX(ME(L) (PROG(K AC FO BO)
  ((ATOM L)(RETURN L))                     Atoms are no macros
  (SETQ K(CAR L))                          First element of given list
  ((EQ K QUOTE)(RETURN L))                 Do not look in quoted lists
  ((EQ K LAMBDA)(GO LL))                   Special treatment for LAMBDA and
  ((EQ K PROG)(GO LL))                     PROG. Skip their list of formals
  (SETQ K(GET(CAR L)(QUOTE RECIPE)))       If recipe, get it. Apply recipe
  (COND(K(RETURN(ME(K(CDR L))))))          to parameters (CDR L). Submit
                                           result to ME again
  ((NULL(CDR L))(GO B))                    One term cannot be infix
  (SETQ K(GET(CADR L)(QUOTE INFIX)))       Get infix property first, if any
  (COND(K(GO INFIX)))                      If so, go to infix (label!)
```

```
B (SETQ K(GET(CAR L)(QUOTE MACRO)))        Get macro property if any
  (COND(K(GO MACRO)))                      If so , go to macro
  (RETURN(CONS(ME(CAR L))(ME(CDR L))))     Otherwise break down recursively
MACRO (SETQ FO(CADR K))                    Formals are second from lambda form
  (SETQ AC(ME(CDR L)))                     Actuals are macro expanded parameters
  (SETQ BO(CADDR K))                       Body is third of lambda form
C (RETURN(ME(COPY BO FO AC)))              Make a copy of the model with
                                           insertion of actuals for formals
                                           and submit result to ME again as
                                           result could contain new macros
LL(RETURN(CONS(CAR L)(CONS(CADR L)         PROG and LAMBDA and the par list
     (ME(CDDR L))))))                      are not expanded but the rest is
INFIX(SETQ FO(CADR L))                     Formals are second of lambda form
  (SETQ AC(LIST(ME(CAR L))(ME(CADDR L))))  Actuals are newly formed list of
                                           first and third, both subjected
                                           to ME themselves
  (SETQ BO(CADDR K))                       Body is third of lambda form
  (COND(CDDDR L)                           If tail not empty copy the infix
    (RETURN(ME(CONS(COPY BO FO AC)         triple, join to tail and submit
       (CDDDR L))))))                      again to ME to look for further
                                           macros
  (GO C) ))                                Otherwise only copy as under C
```

The last step is a function COMPILE combining the macro expansion and the definition of the resulting text.

```
EX(COMPILE(X Y Z) (EX X Y (ME Z)))
```

We can now revisit the little problems of the previous paragraph by defining

```
INFIX(+(A B) PLUS(A B))
INFIX(*(A B) (TIMES A B))  etc. fr - , = , <, and :=
RECIPE(IF (L) (COND((CDDDR L)(LIST COND(LIST(CAR L)(CADDR L))
                                       (LIST T (CADDDDR L))))
                    (T(LIST(CAR L)(CADDR L)))) )
```

This recipe macro will make from (IF A THEN B ELSE C) the form (COND(A B)(T C)) and from (IF A THEN B) the form (A B) for use in PROG's. We shall leave the definition of a fully general IF macro to the reader. We now write

```
COMPILE(TIMES(X Y) (PROG(Z)
  (Z := 0)
  (IF (Y < X) THEN (RETURN (Y * X)))        Still too many brackets in this
  (IF (X = 0) THEN (RETURN Z))              version!
  (X := (X - 1)) (Z := (Z + Y))
  (GO A) ))
```

Of course the macro expander is also usable separately. E.g.
ME((A + B + C)) will give (PLUS(PLUS(A B)C) but
ME((A + (B + C)) will give (PLUS A(PLUS B C))

Another application of recipe macros is for code generation. Suppose that the operation code of a machine is dependent on the data type. We want to write (MUL X) but if X has been declared integer, the instruction (IMUL X) and if X has been declared real, the instruction (FMUL X) has to be generated.

```
EX(DECLARE(X Y) (DEFLIS(LIST(LIST X))Y) )
RECIPE(MUL(X) (COND((GET(CAR X)(QUOTE INT))(LIST(QUOTE IMUL)(CAR X)))
                  ((GET(CAR X)(QUOTE REAL))(LIST(QUOTE FMUL)(CAR X)))
                  (T(PRINT(QUOTE(UNKOWN DATA TYPE)))) ))
```

Now we can give

```
DECLARE(X REAL)
DECLARE(Y INT)
COPMILE(PROGRAM NIL ((MUL X)
                     (MUL Y) )
```

This will compile to

```
(FMUL X)
(IMUL Y)
```

Not worth an extra chapter, but still worth mentioning, especially in a chapter on compilers is the function GENSYM without arguments.
In a compiler it is sometimes necessary to invent a brand new atom (e.g. to label a point for an implicit go to statement of a loop, resulting from a written for statement). (GENSYM) does this for us. It creates a new atom with an identifier different from any other. And even if one would write the same identifier on purpose, it would still be a different atom, as these generated atoms are not placed on the OBLIST.

By lack of space, we shall not give an example of a more complete macro expander, which can also handle the usual priorities of operators.
One of my students once made a nearly full ALGOL 60 compiler in only 6 pages of LISP [76], including own arrays and externally definable procedures.

1.25 The system

A good conjurer will not disclose how his tricks are done. But programming is no conjuring and we shall now give a complete description of the inner processes of the system for running LISP, that we have seen so far. We have seen EVALQUOTE as given in chapter 1.14 and 1.15. But these functions used EVAL and APPLY in their turn.

In contrast to many other languages which are formally defined as far as the syntax goeds, but which are defined in plain English when it comes to the semantics, LISP is formally definable in its own language. By giving the actions of EVAL and APPLY the semantics are formally defined. That is, one can derive the effect from any given program from the definition of EVAL and APPLY. Whether this gives a real understanding of the problem by a human being will remain outside this definition. It is as if one gets a grammar of the Hottentot language, written in Hottentot. When once understanding it, everything is neatly defined. This is exactly what we do in all our everyday dictionaries.

It is a remarkable fact that many mathematicians, who have the name of being analysts, have liked the syntactic definitions and their theory best. But the software engineer, constructivists by definition, often like the analytical method of description by defining a language by its processor. The Vienna definition method comes near to the last one and is strongly based on the ideas found in LISP. In chapter 2 we shall see TRAC, another language defined by its processor and having the capability of self-definition.

This reminds me of a little joke. Is it possible to write a program in LISP so that the result will be exactly that program?
The answer is yes! Here is is:

```
DEFINE(((SELF(LAMBDA(X)(PROG NIL(TERPRI
)(PRINT DEFINE)(PRINT(LIST(LIST(LIST(
QUOTE SELF)(GET(QUOTE SELF)EXPR)))))(
TERPRI)(PRINT(QUOTE SELF))(PRINT(LIST(
QUOTE X))))) ))
SELF(X)
```

The somewhat awkward places at which the lines are broken corresponds exactly to the PRINT function, which tries to fill the lines as much as possible. As far as I know FORTRAN can do a similar program, but ALGOL cannot.

Before we can go into APPLY and EVAL, first a few auxiliary functions. We have seen

```
EX(ASSOC(X A)(COND((NULL A)NIL)
  ((EQ(CAAR A)X)(CAR A))
  (T(ASSOC X(CDR A))) ))
```

to look up the value associated with a name on the association list.

The following function will add new pairs to the association list (or any other list, given as third argument).

```
EX(PAIRLIS(X Y A)(COND((NULL X)A)
  (T(CONS(CONS(CAR X)(CAR Y))(PAIRLIS(CDR X)(CDR Y)A))) ))
```

It will add pairs to the association list given in A to the corresponding elements of the list of formals X and the list of actuals Y. E.g. PAIRLIS((F1 F2)(A1 A2) ((X.3)(F1.5)(Y.7))) will give ((F1.A1)(F2.A2)(X.3)(F1.5)(Y.7)) Now F1 appears twice, but ASSOC will only find the first appearance.

A function to look up properties under a given indicator on the property list of an atom is:

```
EX(GET(OBJ IND) (PROG(K)
  (SETQ K(CDR OBJ))                      Take property list of object in K
  (NULL K)(RETURN NIL))                  Nothing in it
  ((EQ(CAR K)IND)(RETURN(CDR K)))        If found, return with property
  (SETQ K(CDDR K))(GO A) ))              Break down iteratively
```

Now we can give APPLY

```
EX(APPLY(FN ARGS A) (PROG(K)
  ((NULL FN)(RETURN NIL))                NIL as a function returns NIL
  ((FN-pointer < certain value)          Connexion point for the real hard
   (do machine code function))           standard functions in the system
  ((ATOM FN)(GO A))                      Function is given as a name
  ((EQ(CAR FN)FUNARG)                    If FUNARG is met, do APPLY but with
   (RETURN(APPLY(CADR FN)ARGS(CADDR FN)))) the old association list
  ((EQ(CAR FN)LAMBDA)                    If lambda form, associate formals
   (RETURN(EVAL(CADDR FN)(PAIRLIS        with actuals by PAIRLIS and hand
      (CADR FN)ARGS A ))))               over to EVAL
  (RETURN(APPLY(EVAL FN A)ARGS A))       Otherwise evaluate FN first
  (SETQ K(ASSOC FN A))                   Is there an associated value?
  (COND(K(RETURN(APPLY(CDR K)ARGS A))))  If yes, apply it
  (SETQ K(GET FN EXPR))                  Is there an EXPR property
  (COND(K(RETURN(APPLY K ARGS A))))      If so, apply it
  (GO C) ))
```

```
EX(EVAL(FORM A) (PROG(K L)
  ((NULL FORM)(RETURN NIL))                NIL is NIL anywhere
  ((NUMBER FORM)(RETURN FORM))             A number is itself as value
  ((ATOM FORM)(GO A))                      Evaluate atom
B (SETQ K(CAR FORM))
  ((EQ K QUOTE)(RETURN(CADR FORM)))        If QUOTE return with argument itself
  ((EQ K FUNCTI)(RETURN                    If FUNCTION then give arguments
     (LIST FUNARG(CADR FORM)A)))           together with present Alist under
                                           the special indicator FUNARG
  ((EQ K COND)                             Evaluation of COND is left to
   (RETURN(EVCON(CDR FORM)A)))             the function EVCON
  ((EQ K PROC)                             Evaluation of PROG is left to
   (RETURN(EVPROG(CDR FORM)A)))            the function EVPROG
  ((ATOM K)(GO C))                         Other atomic function names
D (RETURN(APPLY K(EVLIS(CDR FORM)A)A))     Otherwise apply the function to
                                           the list of evaluated arguments
                                           This is done by EVLIS
A (SETQ K(ASSOC FORM A))                   Look up form on alist
  (COND(K(RETURN(CDR K)))                  If found, return with value
  (RETURN(GET FORM APVAL))                 Otherwise return with permanent
                                           value from property list
C (SETQ L(GET K EXPR))                     See if K is a EXPR
  (COND(L(RETURN                           If so, then apply the lambda form got
     (APPLY L(EVLIS(CDR FORM)A)A))))       from property to the evaluated
                                           arguments
  (SETQ L(GET K FEXPR))                    See if K is a FEXPR
  (COND(L(RETURN                           If so, then apply the lambda form
     (APPLY L(LIST(CDR FORM)A)A))))        (which should have two formals!)
                                           to the unevaluated arguments and
                                           the alist.
  (RETURN(EVAL(CONS(CDR(ASSOC K A))        If nothing helps, look up function
     (CDR FORM))A)) ))                     in association list and try again
```

The function EVLIS has been given before but we shall give it here again for completeness.

```
EX(EVLIS(L A)(COND((NULL L)NIL)
             (T(CONS(EVAL(CAR L)A)(EVLIS(CDR L)A))) ))
EX(EVCON(C A)(PROG NIL
  ((NULL C)(RETURN NIL))                   No more pairs
  ((NULL(EVAL(CAAR C)A))(GO N))            If first of first pair is not true
                                                                  goto N
  (RETURN(EVAL(CADAR C)A))                 If true, return with value of second
  (SETQ C(CDR C)) (GO L) ))                Otherwise, break down iteratively.
```

Note that with EVCON the conditional form itself is not used. Thanks to the use of T as a function and the sequencing properties of PROG this could be done. Even the few places in APPLY and EVAL where COND is used could be written as ((NULL(NULL ... = ((NOT(NOT ... This will always give T or NIL. Nevertheless, the intensive use of cross recursion of all functions involved gives a somewhat circular flavour to the whole story. At last comes

```
EX(EVPROG(C A)(PROG(K L M)                EVPROG uses PROG !
  (SETQ K(CAR C))                         K is list of prog variables
  (SETQ L(CDR C))                         L is running pointer on C
B ((NULL K)(GO A))                        If no prog var, then goto A
  (SET(CAR K)NIL)                         A rare use of SET! Set all prog vars
  (SETQ K(CDR K))                         to NIL
  (GO B)
A ((NULL L)(RETURN NIL))                  At end of prog list, return NIL
  ((ATOM(CAR L))(GO D))                   Atomic means label. Skip
  (SETQ M(CAR L))                         M is next statement
  ((EQ(CAR M)GO)(GO GO))                  If this is a GO, goto GO
  ((EQ(CAR M)RETURN)(GO RETURN))          If this is a RETURN, goto RETURN
  (EVAL M A)                              Otherwise do the statement and
                                          lose its value. It is "voided" in
                                          ALGOL 68 terminology.
D (SETQ L(CDR L))  (GO A)                 Go to next statement
RETURN(RETURN(EVAL(CDR M)A))              Return with evaluated argument of
                                          RETURN
GO(SETQ M(CDR M))                         M is argument of GO
  (SETQ L C)                              L is a whole prog list again
G ((ATOM(CAR L))(GO E))                   Search for atomic label
H (SETQ L(CDR L)) (GO G)                  Non atomic items are passed
E ((EQ(CAR L)M)(GO D))                    If label looked for is found, resume
  (GO H) ))                               Go on searching.
```

In the whole program above, nothing has been inserted for detecting errors. This has been done deliberately to simplify the presentation. In the real system all kinds of error checks have been built in.

There is another slight deviation from the usual other LISPs in that an associated value is looked up before the permanent property. In fact the implementation with a value cell does not need it. There the property list is completely at the disposal of the user. Functions (EXPR's and FEXPR's) are looked up on the property list first as this is the usual place to find them.

For completeness all remaining functions used but not defined yet will be given here (Except for READ which is to much implementation dependant because of the structure of atoms).

```
FEX(LIST(EVLIS L A))                        LIST is in fact done by EVLIS
FEX(SET(RPLACD(ASSOC(EVA(CAR L)A)A)(EVAL(CADR L)A))))
FEX(SETQ(RPLACD(ASSOC(CAR L)A)(EVAL(CADR L)A))))
EX(NULL(X)(EQ X NIL))
EX(T(X)X)
DEFLIS(((T T))APVAL)
```

All ther standard objects such as EXPR, FEXPR, LAMBDA, APVAL, QUOTE, RETURN, GO etc. can be considered to be defined with themselves as value, so that standard objects need not be quoted (but they may of course!).

A simple form of DEFLIS for defining a single function at a time could be defined as

```
EX(DEFLIS(X Y)(RPLACD(CAAR X)(LIST Y(CDAR X)) ))
```

But the real DEFLIS will look up whether the required indicator is present already. If so, it will replace the present property by a new one, otherwise it will insert a new property together with the new indicator.

```
EX(DEFLIS(L PRO)(MAPCAR L(FUNCTI(LAMBDA(J)
    (DEFA(CAR J)(CADR J)) ))))
EX(DEFA(OBJ L)(PROG NIL
  (RPLACA(PROP OB PRO(FUNCTI(LAMBDA NIL
      (CDDR(RPLACD OB(CONS PRO(CONS NIL(CDR OB)))))  )))L)
  (RETURN OB) ))) ))
EX(PROP(X Y FN)(PROG NIL
  ((NULL X)(RETURN(FN)))
  (EQ(CAR X)Y)(RETURN(CDR X)))
  (SETQ X(CDR X))
  (GO A) ))
```

1.26 A few notes on the implementation of LISP

We shall sketch here the most important primitive functions in an ALGOL 60 imbedded implementation. It is assumed that the store is a sequential array M[low:high] . Each LISP cell shall require two words in this array, the CDR word first and then the CAR word. We furthermore suppose the presence of the boolean procedure odd(x) with an obvious significance.
Nota that all pointers are even. Hence the rightmost bits of both words are available. The bit in the CAR word is used for the atom mark. The bit in the CDR word is used for the collector mark.

```
integer array M[low:high]; comment M[0] contains NIL but is out of the LISP range;
integer procedure CAR(x); value x; integer x;
          if cdd(M[x+1]) then ERROR else CAR := M[x+1] ;
comment As all parameters will be called by value and specified integer, this
  will be omitted in the other procedures;
integer procedure CDR(x); CDR := M[x]
integer procedure CONS(x,y);
begin integer k; k := CELL;
      M[k] := y ; M[k+1] := x; CONS := k end;
integer procedure CELL; comment CELL gives a new cell from a list called FREE;
begin CELL := FREE; FREE := M[FREE]; if FREE = 0 then COLLECTOR end;
integer procedure ATOM(x);
           ATOM := if cdd(M[x+1]) then TRUE else 0 ;
comment ATOM is not a boolean procedure but delivers an integer pointer to TRUE;
integer procedure NUMBER(x);
           NUMBER := if M[x+1] = 1 then TRUE else 0;
integer procedure EQ(x,y);
EQ := if x = y ∨ (NUMBER(x) ≠ 0 ∧ NUMBER(y) ≠ 0 ∧<[<[x]+1] = M[M[y]+1])
      then TRUE else 0;
comment Numbers are supposed to have the number as first element of the
     property list of the atom;
```

The garbage collector COLLECTOR has to preserve all living objects. All living objects are supposed to form a single tree, having its root called STACK. The first object on the STACK shall be the (initially empty) OBLIST. The stack is furthermore used for all recursive actions. We cannot leave this to the recursivity of ALGOL as this stack is inaccessible to the garbage collector. STACK is supposed to be a linked stack, making use of elements from the FREE list when necessary.

```
procedure PUSH(x);
begin integer k;    M[FREE+1] := x;
                    k := M[FREE];
                    M[FREE] := STACK ;
                    STACK := FREE ;
                    FREE := k ;
     if FREE = 0 then COLLECTOR
end;
integer procedure POP;
begin integer k;    POP := M[STACK+1];
                    k := M[STACK];
                    M[STACK] := FREE;
                    FREE := STACK;
                    STACK := k
end  Observe the beautiful symmetry of PUSH and POP. if a rotation instruction
would be available in the hardware, this way of pushing and popping could be
done almost as efficiently as the sequential method of stacking:
procedure COLLECTOR;
begin integer k;    MARK(STACK);
      unmark:       for k := low step 2 until high do
                    if odd(M[k]) then M[k] := M[k] - 1
                                 else begin M[k] := FREE; FREE := k end
end We shall suppose that below low there is a region containing machine code
routines for the primitives. These should not be collected.;
procedure MARK(x);
begin if x < low v odd(M[x]) then goto end ; comment an address reaching into
      the machine code system, or cell, that is already marked finishes the
      marking. A marked cell van be encountered in a circular list;
      M[x] := M[x] + 1;  comment mark this cell;
      if ATOM(x) = TRUE then begin MARK(M[x]-1); MARKATOM(M[x+1] - 1) end
                        else begin MARK(M[x]-1); MARK(CARX)) end ;
end: end  MARK recursively marks CAR and CDR. In the case of an atom, the
      CAR part must be marked with a special routine, as the object under the
      atom header is not conforming to normal list structure. Let us assume
      that the characters are stored in the CAR parts themselves and that the
      CDR's serve to link to the next characters. This is then a list in the
      CDR direction only.;
```

```
procedure MARKATOM(x); value x; integer x;
L:    if x ≠ 0 then begin M[x] := M[x] + 1; x := M[x] - 1; goto L end
      Independent of the structure of the CAR parts, which contain characters,
      the CDR chain is marked iteratively .;
```

It remains to be shown how EVAL and APPLY can be implemented with an explicit STACK and possibly in an environment which is non-recursive.

```
switch S := L1, L2, L3, L4, ....    ; integer n;
EVAL: PUSH(n); PUSH(arg1); ...
            ...
            n := 3; goto EVAL; L3: ...
            ...
            n := 5; goto APPLY; l5: ...
            ...
  ...     ; arg1 := POP; n := POP; goto S[n];
APPLY: PUSH(n); PUSH(arg); etc.
```

Instead of making procedures, the coding is labelled. Every call is made by a goto but before jumping an explicit return n is given, returning to a corresponding Ln: The first action done in any "subroutine" is pushing that return n on the STACK. On exit the same data are popped up in the reverse order in which they were pushed on the stack. The return n will be low enough to stay under the low limit of the array, so that the garbage collector will not MARK any of there return n's. All other objects are pointers, never the values themselves and they are all above the low limit.

This sketch of a possible implementation has been kept as simple as possible. Nevertheless it is sufficient to make an implementation quickly from these almost complete data. It would, however do injustice to the more sophisticated implementations. One of the problems about garbage collection is, that as described here, it is a recursive process in itself. Now a collection is done when all space is consumed. But how much space in the recursion stack will be needed for the collector itself? More advanced schemes do not rely on a recursive marking technique, but they can do a marking by pointer reversing within the pointers themselves and without using any variable stack space. Furthermore, nothing has been said about compacting the cells to one side of the store. This is necessary when a sequential stack is growing from the other side. The problems of doing that compaction are difficult, especially when the elements are of different length as suggested in 1.1. But they have been solved. In a particular implementation we made even compiled machine code is relocated, when shifted during compaction. These results are highly relevant for implementation of ALGOL 68.

References

[1] The Programming Language FORTRAN. ISO Recommendation R 1539. International Organization for standardization. 1971

[2] P. Naur (Ed.). Revised Report on the Algorithmic Language ALGOL 60. International Federation for Information Processing 1962. Numerische Mathematik 4 (1963) 420 - 453.

[3] The Programming Language ALGOL. ISO Recommendation R 1538. International Organization for standardization. 1971

[4] A. Newell (Ed.). Information Processing Language - V Manual. Prentice-Hall, Englewood Cliffs, N.J., 1961

[5] V.H. Yngve. COMIT Programmers' Reference Manual. MIT Press, Cambridge, Mass., 1961.

[6] D.J. Farber, R.E. Griswold, and I.P. Polonsky. The SNOBOL 3 Programming Language. Bell System Technical Journal

[7] R.E. Griswold, J.F. Poage, I.P. Polonsky. The SNOBOL 4 Programming Language. Printice-Hall, Englewood Cliffs, N.J., 1968/

[8] A Forte. SNOBOL 3 Primter. MIT Press, Cambridge, Mass., 1967.

[9] K.E. Iverson. A Programming Language. Wiley, 1962.

[10] Users' Manual APL\360. IBM Document GH20-0683.

[11] Sandra Pakin. APL\360 Reference Manual. Science REsearch Associates, Chicago, Ill., 1968.

[12] E. Bond, M. Auslander, S. Grisoff, R. Kenney, M. Myszewski, J.E. Sammet, R.G. Robey, and S. Zilles. FORMAT, an experimental formula manipulation compiler. Proc. ACM Nat'l Conf. Aug. 1964.

[13] C. Engelman. The MATHLAB Manual. MITRE Corporation, Bedford, Mass. 1971.

[14] A. van Wijgaarden (Ed.). Report on the Algorithmic Language ALGOL 68. Numerische Mathematik 14 (1969) 79 - 218.

[15] C.H. Linsy and S.G. van der Meulen. Informal Introduction to ALGOL 68. North-Holland Publ. Co. Amsterdam, 1971.

[16] J.E.L. Peck. An ALGOL 68 Companion. Dept. of Comp. Sci., Univ. of British Columbia, Vancouver, B.C., Canada, 1972.

[17] P. Branquart, J. Lewi, M. Sintzoff, and P.L. Wodon. The Composition of Semantics in ALGOL 68. Comm. ACM, 14 (1971) 697 - 708.

[18] GPSS, General Purpose Simulation System/360 User's Manual. IBM Document H20-0326.
[19] O.J. Dahl and K. Nygaard. SIMULA, An ALGOL-Based Simulation Language. Comm. ACM, 9 (1966) 671 - 678.
[20] J. Weizenbaum. Symmetric List Processor (SLIP). Comm. ACM. 6 (1963) 524-544.
[21] J.G. Kemeny, and T.E. Kurtz. BASIC Programming. Wiley, New York, 1967.
[22] PL/I Reference Manual. IBM document C28-8201.
[23] N. Wirth. PL360, A Programming Language for the 360 Computers. Journal of the ACM. 15 (1968) 37 - 74.
[24] W.A. Wulf et al. BLISS Reference Manual. Carnegie-Mellon Univ. Comp. Sci. Dept., Pittsburgh, Pa., 1970.
[25] Introduction to Software Engineering with AED-0 Language. SofTech, Waltham, Mass. 1969.
[26] M. Richards. BCPL, A tool for compiler writing and system programming. Proc. Spring Joint Computer Conf.557 - 566.
[27] P.C. Poole, and W.M. Waite. The STAGE2 Macroprocessor user Reference Manual. UK Atomic Energy Authority, Culham Lab., Abingdon Berkshire, 1970.
[28] ALGOL W Programming Manual. Univ. of Newcastle upon Tyne, 1970.
[29] N. Wirth. The Programming Language Pascal. Acta Informatica 1 (1971) 35-63.
[30] R.M. Burstall, J.S. Collins, and R.J. Popplestone. Programming in POP-2. Edinburgh Univ. Press, 1971.
[31] C.N. Mooers. TRAC, A procedure-describing Language for the Reactive Typewriter. Comm. ACM, 9 (1966) 215 - 220.
[32] N. Wirth. EULER: A Generalization of ALGOL and its Formal Definition. Comm. ACM 2 (1966) 13 - 25 and 89 - 99.
[33] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. Comm. ACM. 3 (1960) Apr.
[34] J. McCarthy et. al. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass. 1965.
[35] E.C. Berkeley, and D.G. Bobrow (Editors). The Programming Language LISP, Its Operation and Applications. MIT Press, Cambridge, Mass. 1966.
[36] (LISP 1.5 PRIMER(BY(CLARK WEISSMAN))). Dickenson Publ. Co., Belmont, Cal. 1967.

[37] D. Knuth. Man or Boy? ALGOL Bulletin 17.2.4., 1964.

[38] C. Strachey. A General Purpose Macro Generator. The Computer Journal, 8 (1965), 255 - 241.

[39] J.W. Mevius. HILT, een programmeertaal gebaseerd op TRAC< Master Thesis, Univ. of Delft, 1971.

[40] J. Moses. The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem. SIGSAM Bulletin of the ACM, No. 15, July 1970, 13 - 27.

[41] E. Sandewall. A Proposed Solution to the FUNARG Problem. SIGSAW Bulletin No. 17, Jan 1971, 29 - 42.

[42] S.J. Houghton. Design and Implementation of an Extended List Processor. Doctoral Thesis, Univ. of Bradford, England, June 1971.

[43] J. Weizenbaum. Recovery of Reentrant List Structurs in SLIP. Comm. ACM 12 (1969) 370 - 372.

[44] M.V. Wilkes. An Experiment with a Self-Compiling Compiler for a Simple Lisp-Processing Language. Annual Review in Automatic Programming, Vol. 4. Pergamon Press, 1964.

[45] J.W. de Bakker. Recursive Procedures. Mathematical Centre Tracts, No. 24. Mathematisch Centrum Amsterdam, 1971.

[46] R. Leentfaar. Een LISP-compiler voor ALGOL. Master Thesis, Univ. of Delft, 1968.

[47] M. l. Minsky. Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, N. J., 1967.