MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

AI-199

MAC-M-428
June 1970

The Function of FUNCTION in LISP

or

Why the FUNARG Problem Should be Called the Environment Problem

by

Joel Moses

Abstract

A problem common to many powerful programming languages arises when one has to determine what values to assign to free variables in functions. Different implementational approaches which attempt to solve the problem are considered. The discussion concentrates on LISP implementations and points out why most current LISP systems are not as general as the original LISP 1.5 system. Readers not familiar with LISP should be able to read this paper without difficulty since we have tried to couch the argument in ALGOL-like terms as much as possible.

Most computer programmers are familiar with the use of a stack for maintaining values for arguments and local variables in recursive functions. We shall assume, for concreteness, that there is an index register which contains a pointer to the end of the stack, called a stack pointer. Variable values and other information in the stack can thus be accessed as small decrements or increments (depending upon the machine implementation) to the stack pointer's value. We shall further assume that a stack grows downward whenever a function is entered. Thus, in Figure 1 we see the stack pointer just before entering a function f, immediately after entering f with argument x and local variable a, and finally immediately after f has returned a value.

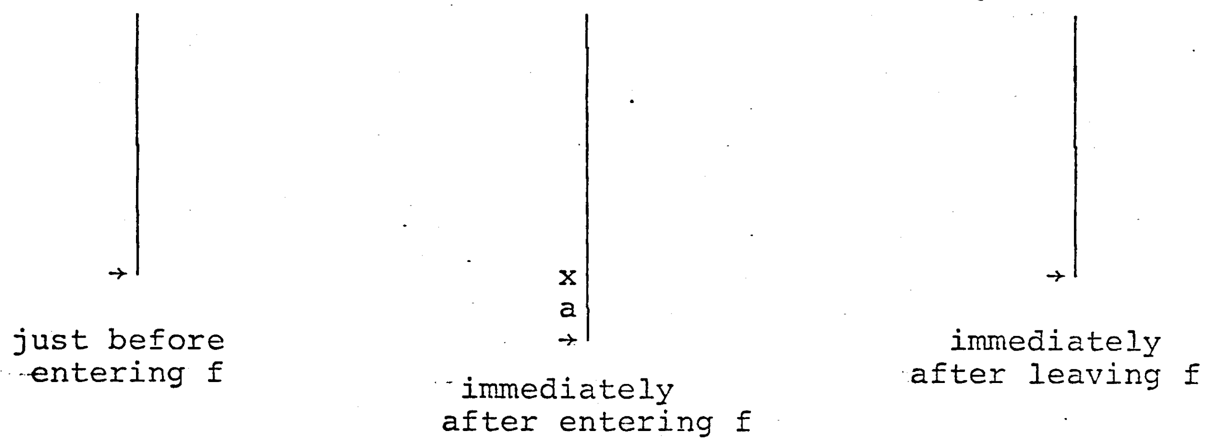


Figure 1

Using the stack mechanism it is, as we have noted, quite easy to obtain the value of any argument and local variable. The situation is more complex if we wanted to provide for free variables. A variable is used free in a function if it is neither a formal argument nor local to that function. For example, in the function f given by

```
define f(x) =
   begin;
      if a>0 then return x
         else return -x;
            end;
```

, the variable _a_ is used free. Of course for the function to be meaningful the variable _a_ must be bound (that is, not free) in some function or block from which f is called, or which calls g which calls f, etc.

The use of free variables is a very powerful computational idea. In many situations it is possible to avoid using free variables by supplying a function with additional arguments. This can be quite cumbersome. It is also inefficient since a subroutine call might require the saving of dozens of parameters. The idea of substituting the values of the free variables inside the functions is also very inefficient since, for example, one must differentiate between occurrences of a variable which are free and those which are not. In addition, both of these methods do not allow one to modify the value of the free variable and achieve the desired effect.

When a function uses free variables or when it calls functions which use them, then the vlaue of this function is not completely determined by its arguments, since the value might depend on the values of the free variables. Thus in order to completely specify a computation, one has to specify a function and its arguments in addition to an _environment_ in which the values of the free variables can be determined.[1] A stack is one way of maintaining such environments. The goal of this paper is to show why great care must be used in maintaining environments.

1. Inputs from external sources may, of course, effect the value of a computation. We shall consider input values to be part of the environment also.

We shall do this by indicating several situations in which it becomes increasingly complex to keep the proper environment needed for further computation.

Probably the simplest situation in which free variables occur is in a block-structured program. We assume here a block (in the Algol sense) which has one or more blocks local to it, where each of these blocks may contain local blocks, etc. A variable which accurs free in the inner blocks can be accessed and modified with ease since the compiler knows where it has stored the value for that free variable on the stack. Thus the free variable a in the program segment below is easily accessed because it is used lexically near the place where it is bound.

```
                      begin local a;
  'a'                    a = 2;
 bound                begin local x;
          'a'            x = f(2) + a;
                         :
          free         end;
                      end;
```

Consider, however, the situation in which the free variable is used inside of a recursive function. Now it is difficult to know where in the stack its value is stored even though the variable is bound lexically near the function call.

```
        define g(x) =
  'a'      begin;
 free         :
              a
              :
              g
              :
        end;
        begin local a,b;
  'a'      a = 5;
bound      b = g(2);
        end;
```

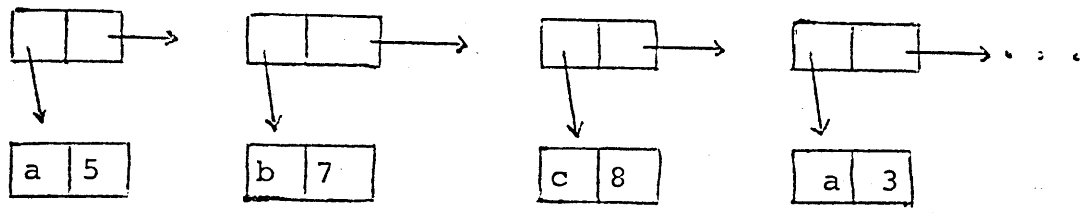
There are several ways in which to maintain the values of free variables (and, thus the computational environment) in order to handle cases like the one above. All of the techniques used involve some increase in time. One approach makes it easy to access the value of a free variable while increasing the cost of entering and leaving a block in which such a variable is local. This approach is favored in recent implementations of LISP. We shall call it the "shallow access" approach.

Another approach is to make it relatively easy to enter and leave a block in which a free variable is bound, but relatively expensive to access a free variable. This approach is used in the classical "alist" implementation of LISP. Several Algol systems have opted for a similar approach. We shall call this the "deep access" approach. Both of these approaches allow one to modify the value of the free variable as well as access it. The access time is approximately the same as the modification time in both approaches.

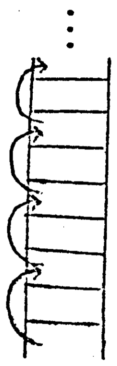
Let us consider the "shallow access" approach first. By "shallow access" we mean that the value of a free variable can be obtained in a single fetch from memory. Since the current value of the free variable may be stored in a difficult-to-determine location up in the stack, a special cell for its current value is used. In many recent LISP implementations this special value cell is stored as a property of the atom structure, and is unique to the free variable. In order to maintain the proper value in the special cell, extra work must be done every time a function or a block is entered in which the variable is an argument or is local. On entering such a function or block one usually saves the old value which is in the special cell on the stack along with sufficient information to allow one to know from where the stack value came. On leaving such a function or block one must remember to store the value saved in the stack in the special cell.

The "deep access" approach forces one to search upward in the stack for the most recent value of the free variable. Such a search may be quite deep and slow if the free variable was bound very far up in the stack. However, this approach has the advantage of avoiding the need for an extra special value cell. In addition, we shall see later that this approach sometimes allows one to use free variables with greater ease and flexibility than the "shallow access" approach.

The LISP "alist" implementation is, as we noted earlier, an example of the "deep access" approach. The "alist" contains pointers to previous values of all bound variables together with the variable names. The name of the variable appears on the left side of a pair, and the value of the variable on the right side. Thus the first variable encountered below is 'a', and its value is 5.



In some Algol implementations, all variables local to a given block are bound on consecutive locations in a linear stack. In addition there are pointers in the stack which allow one to get from the values local to one block to those local to higher blocks. Thus, if one is forced to search for the value of a free variable, one can use the pointers to get from values local to one block to those in another, until one arrives at the block at which the free variable is bound.



The real advantage for "deep access" is seen when one considers passing functions from one block or function to another. We shall first consider the easy case when a function is passed downward - that is, when it is used as an argument to another function. The hard case, when a function is returned as the value of a computation, will be considered later. It should be noted that the hard case does not occur in ALGOL 60 or ALGOL 68 because returns of functions are not allowed in these languages.

Let us recall that the evaluation of a function is dependent on its arguments plus the environment which gives meaning to any free variables used by it or its subroutines. An important point which must be realized about functional arguments (abbreviated FUNARGs) is that two different environments are involved in such cases. The first environment is the one which is in effect when the FUNARG is bound as an argument. We shall call this the <u>binding environment</u>. The second environment is the one that is in effect when the FUNARG is activated as a function. We shall call this the <u>activation environment</u>.

Since the binding environment and activation environment will, in general, differ from each other, it is a nontrivial matter to decide which environment to use in order to evaluate a functional argument. Consider the following example:

```
                          define f(x) =
                             begin;
                             if a = 0 then return x
                                else return -x; end;

                          define g(x,fun) =
                             begin local a;
                                a = 0
    activation environment      return x + fun(x); end;

                          begin local a,b;
                             a = 1
    binding environment      b = g(3,f);
                             end;
```

Note that the binding environment has a = 1, and that the activation environment has a = 0. If we used the binding environment to evaluate f(x), then its value would have been -3. If we had used the activation environment to evaluate f(x), then its value would have been 3. Thus the importance of determining which environment to use should be clear.

From an implementational point of view, it is obvious which environment one would like to give a functional argument - it is the activation environment. The reason for this is that in the "shallow access" implementation the values of the free variables are then set correctly. In the "deep access" case, the values of the free variables are close at hand. Unfortunately the programmer would usually like to have the free variables have the value they possess in the binding environment.

Consider now what it would require for the system to restore the binding environment for functional arguments. It would require knowing where in the stack or "alist" the binding environment exists through some pointer to it. Supplying such a pointer is a function of FUNCTION in LISP. That is when one transmits a functional argument f which is to be evaluated in its binding environment, then one uses FUNCTION(f) instead of QUOTE(f). FUNCTION will prevent its argument f from being evaluated, just as QUOTE would. The result of FUNCTION will be a structure which not only contains a reference to the function f, but also contains a pointer to the binding environment. Thus at the FUNARG's activation time we will be able to use the pointer to restore the environment to the proper place. In the "deep access" case with a stack implementation the restoration process is not too costly because one creates a pointer from the current place in the stack up to the point where the binding environment exists, thus skipping over any values in the activation environment which differ from those in the binding environment. (see Figure 2.)

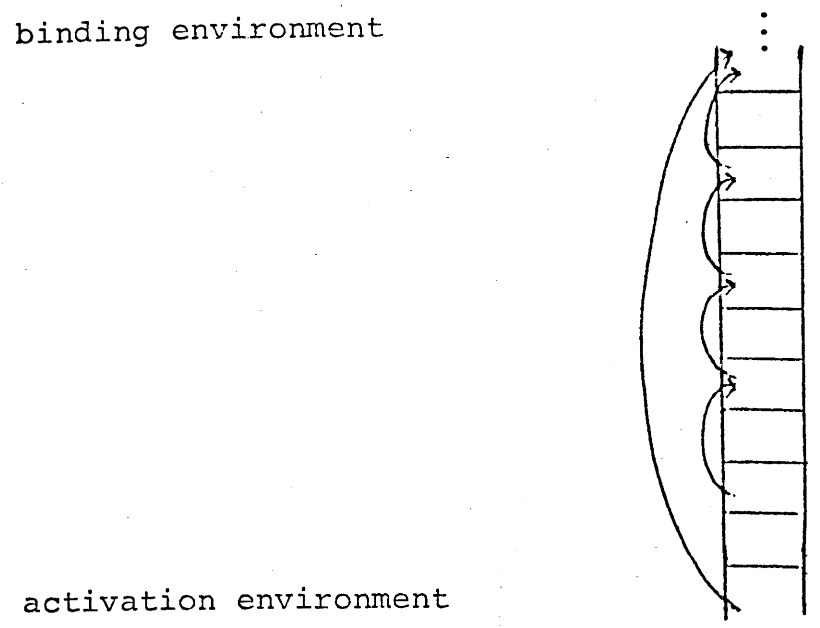


Figure 2.

In the "alist" implementation this process is even easier, because the saved pointer is made the current "alist", and we are done because the values of free variables will be obtained from the "alist" which contains the binding environment.

In the "shallow access" case, the matter is much more complicated. One has to restore the special value cells by going upward in the stack and giving the free variables their previous bindings. This is done until one gets all the way back to the binding environment. After the functional argument has been evaluated, the environment must be restored to the activation environment. (There can be no question about the necessity of this change, of course.) In the "deep access" case the stack is automatically restored to the correct environment. In the "shallow access" case one must repeat the restoration process, only now one performs it downward from the binding environment to the activation environment.

It might seem that this change in environments for functional arguments causes enough implementational difficulties for the "shallow access" school that there could not possibly be any more complications. In fact some implementers of "shallow access"

LISP systems who have gone as far as we have described have thought that they had solved all the problems related to maintaining the proper environment (see Bobrow, et al. [1]). However, we pointed out earlier that the worst case is yet to come. This is the case where the function is returned as the result of some computation.

This case is difficult for the "shallow access" approach because one would normally discard the portion of the stack through which the return was made. The space occupied by the discarded part of the stack will then be usable in subsequent computations. However, if we are to preserve the binding environment, then we cannot discard the portion of the stack through which the return was made.

Consider what might happen if we did discard values in the binding environment in the example below.

```
define f(x) =
   begin;
      if a = 0 then return x
      else return -x;
      end;
define g(x) =
   begin local a;
      a = 2;
   return f                                  binding environment
      end;
begin local a,b,h;
   a = 0;
   h = g(2);
   b = h(3);                                 activation environment
   end;
```

In the example above the value of a at binding time in g is 2 and at activation time it is 0. If we had discarded the environment generated by calling g, then we would surely have lost the value of 2 for a. In fact, if we had tried not to discard the stack and had saved a pointer to the binding environment, we would find a lot of garbage there created by the call to f.

Therefore, when one returns a function which is to be activated with its binding environment, one must save the binding environment so that one can get back into it. But how does one restore the binding environment? One has to essentially back-up the stack until the point is reached where the function was returned. But that is not enough since one must then back-down through the saved environment to reach the point where the binding was made and the return initiated (see Figure 3).

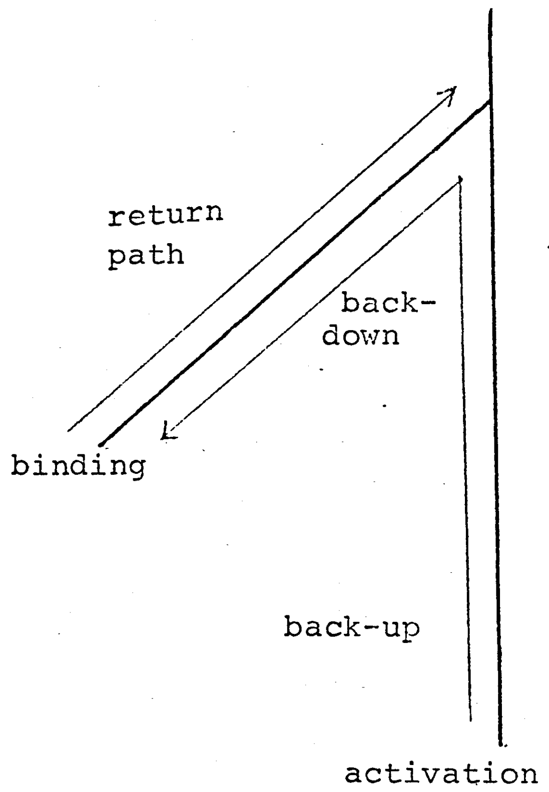


Figure 3

We see that by not discarding the stack during the return we have created an environment with a branch. If we allow several function returns without a discard, then the environment we have created is a tree. The question now is how does one create a tree structure from what was originally a linear stack. If we allow both upward pointers and downward pointers in the stack, then we can have a tree. So the "deep access" implementers with upward pointers need only introduce downward pointers and their problems are solved.[2] The "deep access" implementers with an "alist" always had a tree (though many of them probably did not realize its importance), and thus they encounter no additional problems in this case. However, woe to the "shallow access" implementers. They must maintain the binding environment and not accidentally obtain variable values from it. They must also back-up and then back-down the stack in order to reach the binding environment. Finally, they must reverse the process upon leaving the returned function. We know of no LISP implementation with "shallow access" which attempts to solve this case of the "environment problem."[3]

2. Unfortunately, there are difficulties in this implementation in reclaiming discarded parts of the stack. A solution which minimizes the amount of storage required for maintaining an environment is to abandon a stack and use a structure which is quite close to an "alist" (see Figure 4.)

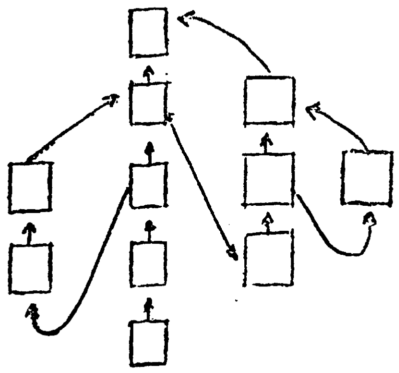


Figure 4.

Such a structure which is composed of noncontiguous linear arrays connected via upward and downward pointers can be automatically garbage collected by checking to see if pointers exist to substructures.

3. It should be clear at this point that the FUNARG problem is badly named because the difficulties are not completely solved when one can handle functional arguments. One should also handle returns of functions, or what is essentially the same problem - the case when a free variable is assigned a function as value. For this reason we prefer the term "Environment Problem."

The points we have made so far are:
1) Free variables in function definitions require that one must have an environment in order to be able to evaluate a function.
2) When one uses functional arguments, then the question arises as to which environment one chooses in order to evaluate the function, the binding environment or the activation environment.
3) When one returns functions (which require the values of free variables for their evaluation) then one must, in general, save the binding environment and thus create a stack which is, in fact, a tree.

Some language designers have made the unfortunate decision that all functions are evaluated against the binding environment. LISP allows the user to choose between the binding environment and the activation environment by allowing a function to be bound with FUNCTION or with QUOTE. QUOTE will force the activation environment to be used in evaluating the function. A useful metaphor for the difference between FUNCTION and QUOTE in LISP is to think of QUOTE as a porpous or an open covering of the function since free variables escape to the current environment. FUNCTION acts as a closed or nonporous covering (hence the term "closure" used by Landin). Thus we talk of "open" Lambda expressions (functions in LISP are usually Lambda expressions) and "closed" Lambda expressions. It has been our experience that most LISP programmers rarely return closed Lambda expressions, and hence rarely encounter the full environment problem. There does not currently appear to be any way of solving the complete environment problem efficiently without losing efficiency whenever one accesses or modifies free variables. It is also a complex task now to design a system which offers "shallow access" at some times and "deep access" at other times.[4] As long as these implementational difficulties remain, designers will be justified in providing only incomplete solutions of the environment problem.

4. Certain LISP systems use "deep access" in interpreted functions and "shallow access" in compiled functions. If free variables in compiled functions are declared to be COMMON, their values will be stored on the "alist". In this manner one can solve the environment problem. The cost is a very slow interpreter and "deep access" to free variables.

Historical Perspectives and Acknowledgments

Peter Landin once said that most papers in Computer Science describe how their author learned what someone else already knew. This paper is no exception to that rule. My interest in the environment problem began while Landin, who had a deep understanding of the problem, visited MIT during 1966-67. I then realized the correspondence between the FUNARG lists which are the results of the evaluation of "closed" Lambda expressions in LISP[2] and ISWIM's Lambda Closures [3,4]. Joseph Weizenbaum got sufficiently interested in Landin's work that he implemented a subset of ISWIM based on his SLIP-OPL system. The tree structure of the resulting stack was made quite vivid to me when Weizenbaum encountered difficulty in implementing the backward pointers necessary in the general case. The reason for this difficulty is that one could not then garbage collect the circular SLIP structures which are created in a straight-forward implementation. This is what motivated Weizenbaum to finally introduce classical garbage collection into SLIP [5].

The first version of this paper was written in 1967 when I became incensed at the fact that many LISP implementers did not understand or even care about the problem. Subsequently, Professor Weizenbaum wrote a very readable account of the problem and its solution in his implementation [6]. The present version of this paper owes much of its emphasis on implementations to Weizenbaum's account.

I also wish to acknowledge useful discussions with Professor Robert Fenichel and Carl Hewitt of MIT.

References

1) Daniel G. Bobrow and Daniel L. Murphy, "Structure of a LISP System Using Two-level Storage", CACM, vol.10, no.3, March 1967, pp. 155-159. (Note especially the footnote on p.158.)

2) J. McCarthy et al., LISP 1.5 Programmer's Manual, MIT Press, 1965. (Note especially pp.70-71).

3) P.J. Landin, "The Next 700 Programming Languages", CACM, vol.9, no.3, March 1966, pp.157-165.

4) P.J. Landin, "A λ-Calculus Approach", in Advances in Programming and Non-Numerical Computation, Pegamon Press, 1966, pp.77-141.

5) J. Weizenbaum, "Recovery of Reentrant List Structures in SLIP", CACM, vol.12, no.7, July 1969, pp.370-372.

6) J. Weizenbaum, "The FUNARG Problem Explained", unpublished memorandum, MIT, 1968.