THE FUNARG PROBLEM EXPLAINED

by

Joseph Weizenbaum

Massachusetts Institute of Technology
Cambridge, Massachusetts

March, 1968

Abstract

The FUNARG problem arises in languages that permit functions to produce functions as their values where the produced functions have free variables. This paper shows that a symbol table tree (in place of a stack) serves to avoid identifier collisions appropriately. The paper develops a lambda calculus argument as a tutorial for the problem and its solution.

Work reported herein was supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

In an important sense, this paper is about a single problem -- the so-called FUNARG problem that first arose in LISP. The original LISP implementation solved that problem. Still, years after that implementation, the problem remains ill understood. Bobrow, for example, published a faulty solution for it only recently. (See Bobrow, 1967.)

Fundamentally the problem is this: suppose we have an interpreter that allows the definition of functions. Some functions are executed (i.e., applied to arguments), for their effect and others for the value they leave in an intermediate result register or perhaps on an intermediate result list. We normally think of functions as leaving numbers, strings, lists, or arrays as their values. But it is possible to define functions that leave functions as their values. We may, for example, write

$$F(X) = G \text{ where } G(y) = X^2 + y^2$$

in ordinary mathematical notation. If we then defined

$$P = F(3)$$

and computed

$$P(4)$$

we would expect to get 25 as a result. To define the same function in a program, a programmer might write:

```
DEFINE F(X) TO BE
   LET G(Y) BE
      RETURN X ↑ 2 + Y ↑ 2
   END
   RETURN G
END
```

and at some subsequent point in the program

$$P \leftarrow F(3).$$

After this assignment to P, P designates a function, namely the function returned by F. Our system must somehow remember that P is the value of F applied to an argument that evaluated to 3. Or, to put it another way, the function designated by P, i.e. $X^2 + y^2$, contains X as a free variable but, because of the way P was formed, X is already bound (to 3) on a higher level. This binding must be remembered. Furthermore, the programmer using the function F should not have to know what identifiers were used in its construction. The outcome of F should be unchanged if, for example, every "X" in it has been replaced by an "S" and every "Y" by a "T". We must therefore allow for the possibility that the programmer used identifiers in his program that also appear in the body of F. Hence protection against identifier collisions, that is confusion between two distinct variables having the same name, must be built into the system.

We show that the solution to the problem lies in abandoning the usual symbol table stack for a symbol table tree.

We unfold our argument by the incremental development of a programming language. We initially assume that the interpreter for that language operates with a stack like symbol table structure. When, as we add the facility to define function producing functions where the produced functions have free variables, the stack mechanism proves inadequate, we generalize it to a tree structured symbol table. Along the way, we must develop a fairly careful statement of what a function in our system is. We appeal to relevant parts of the λ calculus to give us a canonical notation.

The idea that should finally stick with the reader is that functions are applied to their arguments in specific contexts -- here called environments. In a way, functions are programs and the environments in which they may be exercised their data. Most programming systems hide this fact

in that they permit functions defined within them to have no free variables and to be applied only in implicitly generated environments. The system shown here permits the application of functions in arbitrary environments.

Suppose the system accumulates statements typed by the programmer and forms them into a program. When the statement "EXECUTE" is entered, the system executes the program accumulated up to that time. Suppose also that programs, once executed, disappear, but that variables and the values they have been assigned are not ordinarily erased.

If, for example, a programmer were to type:

```
A ← 5
X ← A + 3
TYPE X * X
EXECUTE
```

the external response of the system would be to type

```
X * X = 64
```

and internally a symbol table would have been augmented with the identifier 'A' having the value '5' and an identifier 'X' having the value '8'. If the programmer were then to write

```
TYPE A, X
EXECUTE
```

the machine response would be

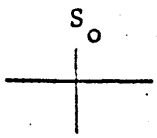
```
A = 5
X = 8
```

thus demonstrating that A and X and their values had been remembered.

The component expressions of every individual statement of a program are evaluated (if at all) in an environment, i.e. with respect to a data structure that provides places for identifiers and the values associated

with the variables the identifiers represent. In the simplest case an environment is a single symbol table. More generally, it is a chain of symbol tables that, in a certain sense, appear and disappear during the execution of a program. One member of such a chain must always be exceptional in that it is the symbol table into which new variables and their values are placed and that in which the search (table look up) for values of variables begins. We shall refer to that exceptional symbol table as the regnant symbol table, or RST for short. It will prove convenient to assign a name to each symbol table we have accasion to mention. We will use a subscripted "S" to denote such names.

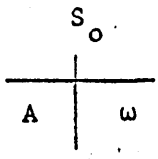
For the program shown above the initital environment consisted of a single empty symbol table S_o



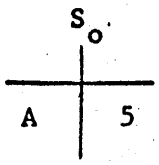
When the assignment statement

$$A \leftarrow 5$$

is encountered and recognized as an assignment statement, the environment is searched for the identifier "A". Since it cannot be found, it is added to the RST with the value "ω" (to be read "undefined") and, in effect, the location of the newly placed "ω" placed on an intermediate result list (IRL). S_o is then:



After the right-hand side of the assignment statement has been evaluated (and 5 pushed unto IRL), the assignment is completed by suitably popping IRL and modifying S_o to be



Had "A" already existed on S_o, the machine address of its value would have been placed on IRL at the appropriate time and the same assignment mechanism invoked in all other respects. Details of the evaluation of expression are not relevant to the present discussion.

But the fact that identifiers (or some token for them, the distinction isn't important here) appear in expressions to be evaluated calls for a word about access mechanisms to values of variables. Clearly, when in the simple system so far discussed, the value of a variable is required, a simple table look up is sufficient. Things will get more interesting soon.

A function definition facility is now added to the system. The form

```
DEFINE F(X) TO BE
    RETURN X + 1
END
```

is intended to define the successor function and add it to the system. What is desired is that a variable F be given some structure as a value, a structure that contains the information

1) that it represents a function
2) that the formal parameter of the function is X

and 3) that the body of the function is "X + 1"

This information could be packaged in a list as follows:

(the function of X that is X + 1)

and that list (or, more precisely, its name) given as a value to F. But that notation is rather long winded. It is shorter as well as more conventional to make the substitutions

"λ" for "The function of"

and

"." for "that is".

The value of F then becomes

(λ X. X + 1)

In general, the form of a lambda expression (as the above is normally called) is

a left parenthesis

followed by "λ"

followed by the formal parameters of the function being defined

followed by a period

followed by the body of the function

followed by a right parenthesis

Let us now follow what happens when we try to evaluate, say F(A), starting with an S_o that is

S_0

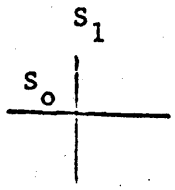
A	3
X	8
F	(λ. X. X + 1)

The interpreter encounters the term F(A) and must prepare itself to apply the function F to the current value of A. We may assume that the

evaluation of A has left 3 on IRL and that the interpreter will execute the program "X + 1". Since X is the formal parameter of F, an association between that X and 3 must be established. On the other hand, the value associated with the X already in the symbol table must not be disturbed. An identifier collision is avoided by the following steps initiated after A has been evaluated and the value of F, in this case the lambda expression "(λ X. X + 1)", found:

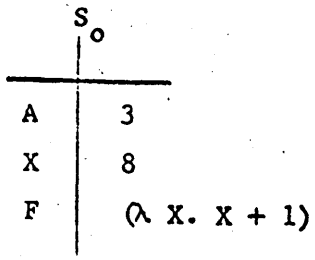
1) The name of the currently regnant symbol table (in this case S_o) is stored.

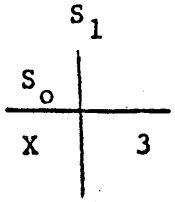
2) A new symbol table S_1 is created and the name of the formerly regnant symbol table placed in a special place on it:



3) The parameter list of the function about to be invoked is copied into the new symbol table and values suitably popped off IRL assigned to these variables.

The finding and placing rules for values of variables must now be enhanced. Recall that just before F is applied to 3, in our example, the symbol table is:





S_1 is now the regnant symbol table. When an identifier is now sought, the search begins in the RST. If it is not found there, the next level (as given by the datum stored in the NW quadrant in our representation) is searched, then the next, and so on. A search may be initiated either for purposes of subsequent value assignment or simply for the retrieval of the value of a variable. If, in the latter case, the appropriate identifier is not found, an error has occurred and the case is not longer of interest here. We choose, when making assignments, to place the identifier to be paired with "ω" in the regnant environment in the manner already described in case of failure to find the identifier.

The application of F to A may now proceed without fear of identifier collision. Upon completion of the application, i.e., once F(A) has been computed, 4 will have been left on IRL. Now the symbol table situation must be restored.

We may ask whether the current RST is the one that was regnant when the function F was first invoked. If yes, then the situation is restored, otherwise we restore the symbol table antecedent to the current ST, throwing away the then regnant symbol table, then enter the restoration process again, and so on. Finally, the original environment will have been restored.

Lst's take another look at the general form

DEFINE name of function (parameter list) TO BE

body of function

END

We see that, when viewed operationally, it is really a fancy form of assignment statement. For the name of the function is really nothing more nor less than an ordinary identifier. The remaining information content of the DEFINE statement is ultimately encoded in the form of a λ expression. The programmer may, in other words, just as well have written

$$F \leftarrow (\lambda\ X.\ X + 1)$$

in place of the definition shown earlier. Perhaps the language we are developing here should have an explicit DEFINE statement. If so, the only justification for its existence can be that that form is somehow prettier than the straightforward assignment of a λ expression to a variable. Landin calls that sort of thing "syntactic sugar".

An important consequence of the fact that a function can be defined by means of a simple assignment statement is that the variable to which the function is assigned as a value is an ordinary variable. It is, for example, subject to reassignment. The λ expression itself is an ordinary datum, e.g., it is subject to being passed around, say as a parameter to a function, or left as the value of another function, just as in any other datum, say a number.

The body of a function defined by the programmer is, of course, a block nested within the block in which the function definition itself is made. One operational function of blocks is to resolve identifier scope questions of the kind that arose in the above example. There, care had to be taken that the bound variable X be given scope within the block, but

not outside it, i.e., that no confusion be permitted to arise between the X with scope inside the block and any X that might have existed before the call to the function F(X).

Of course, the function (λ X. X + 1) is very simple. A somewhat more complicated one that will serve to raise some additional points is defined by

```
DEFINE G(Z) TO BE
    Q ← P /(1-P)
    RETURN (Z + Q) / 2
END
```

The only variable that appeared in the function F shown earlier was X. But in the body of G shown above, the variables P, Q and Z occur and, of these, only Z is bound and can therefore be handled by the techniques already discussed.

Let us assume that P is intended to be a variable that, from the point of view of G, has global scope, i.e., that by the time G is called some environment containing P and its value will exist. Q may similarly exist in some higher symbol table at that time but the programmer may have intended Q to be local to G. Let's suppose so. Then clearly, he did not intend for the value of any previously existing Q to be disturbed by any assignment of the local Q. The mechanism we have already seen clearly makes the choice of names for the bound variables of functions (in the present case "Z") irrelevant. The function G would work just as well had the programmer written "S" where ever he in fact wrote "Z". The same circumstance should pertain to "Q". There should, in other words, exist a mechanism that can declare a variable (or variables) local to a specific block and (although this is not logically necessary) give it an initial value.

Recall now that when an assignment is to be made the system searches the environment, beginning with the regnant symbol table, for (essentially) the position of the then existing value of the most recently established variable bearing the name of the variable that is the object of the current assignment. If it finds one such, then its value is replaced, otherwise a new position for the relevant identifier is found in the regnant symbol table. What we require now is a notation that will inform the system that an assignment statement so denoted should simply add the relevant identifier and the value of the variable it represents to then regnant symbol table, skipping the otherwise usual search. We will indicate that this kind of assignment is meant when we write

"LET $\alpha \leftarrow$ expression".

We again supply a little syntactic sugar when we allow

```
LET F(X) BE
   body of function
END
```

to be written in place of

LET F $\leftarrow$ (λ X. body of function).

Put most simply, the difference between an ordinary and a LET assignment is that the first may reassign the value of a variable originally established in an earlier environment while the latter always modifies the environment regnant at the time of its execution. That is, of course, also the only difference between DEFINE and LET as applied to function definitions.

We now rewrite the definition of the function shown above as follows:

```
LET G(Z) BE
   LET Q ← P / (1-P)
   RETURN (Z + Q) / 2
END
```

Let's assume that, when G is called the regnant symbol table is

	S_i
S_{i-1}	
P	.25
Q	.5
G	(λ Z. Z + Q)

Now the statement "T ← G(1)" is encountered. Just before the system exists from G, the environment will be

	⋮
	S_i
S_{i-1}	
P	.25
Q	.5
G	(λ Z. Z + Q)
T	ω

	S_{i+1}
S_i	
Z	1
Q	.33

and after the assignment to T is completed:

S_{i-1} \ S_i	⋮
P	.25
Q	.5
G	(λ Z. Z+1)
T	.6¢

i.e., the "Q" local to G is gone along with the "Z" that was a formal parameter of G.

We remarked earlier that the choice of the name "Q" for the local variable we had in mind in the above example was completely arbitrary. In mathematics we might write, for example

$$G(Z) = (Z + Q) / 2 \text{ where } Q = P / (1-P)$$

and in that notation as well, the choice of names for what are essentially dummy variables, i.e., Z and Q, is arbitrary. (We could, by the way, have introduced a WHERE notation that would be entirely equivalent to the LET notation we have. WHERE statements would always be written at the bottom of blocks to which they apply. The interpreter would have to inspect each block, as the block is entered, for the presence of a WHERE statement and execute it if one were found. There is, of course, no reason that either a LET or a WHERE statement could not make multiple assignments.) Consider then how we might write

$$H(X) = F(X) + F(X+1) \text{ where } F(Z) = Z^2$$

in our language.

We write:

```
LET H(X) BE
   LET F(Z) BE
      RETURN Z ↑ 2
   END
   RETURN F(X) + F(X+1)
END
```

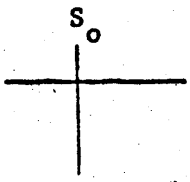
Nothing unusual has been introduced here and the reader should therefore have no trouble in simulating the machine when, say, H(3) is called. Let's introduce a small change in H now and follow what happens. We redefine the above function as follows:

$$H(X) = F \text{ where } F(Z) = Z^2.$$

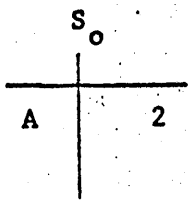
H is thus a function which produces a function as its value. Let's, for the moment, be tolerant of the fact that the whole exercise appears trivial and follow the course of the program shown below nevertheless.

```
A ← 2
LET H(X) BE
   LET F(Z) BE
      RETURN Z ↑ 2
   END
   RETURN F
END
C ← H(0)
D ← C(3)
```

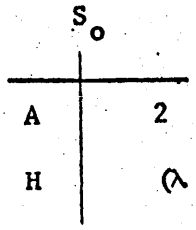
We display the symbol table at each stage



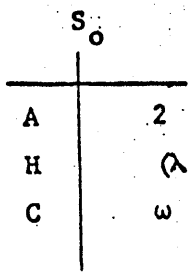
initial stage



after the assignment to A



H has been defined. ";" is a separation between statements in H.



S_1

S_o	
X	0
F	(λ Z. Z ↑ 2)

H has been entered (a new environment node is therefore created) and the LET statement within H executed. "(λ Z. Z ↑ 2)" is now on IRL but we have not yet exited from H.

S_o

A	2
H	(λ X. ...)
C	(λ Z. Z ↑ 2)

The assignment to C has been made.

S_o

A	2
H	(λ X. ...)
C	(λ Z. Z ↑ 2)
D	ω

"D" has been placed on S_o preparatory to assignment.

$S_{\bar{o}}$

A	2
H	(λ X. ...)
C	(λ Z. Z ↑ 2)
D	ω

S_2

S_o	
Z	3

C(3) has been called, the value of C, i.e., (λ Z. Z ↑ 2), found and the formal parameter Z bound to 3 in a newly established RST.

S_o

A	2
H	(λ X. ...)
C	(λ Z. Z ↑ 2)
D	9

The application of C to 3 is completed.

We have here seen the application of a function (H) to a parameter (0) in which the function produced another function (λ Z. Z ↑ 2) as its value. That latter function was assigned (to C) and subsequently applied to a specific argument (3). What made the entire exercise appear trivial was that the argument given to H was entirely irrelevant to the outcome of H, in this case to the function produced by H. Had we written

```
LET H(X) BE
    IF X < 0 THEN 0 ELSE
        LET F(Z) BE
            RETURN Z ↑ 2
        END
        RETURN F
END
```

Then the outcome of the application of H to an argument would have been different depending on the sign of the argument. But that's still not very interesting.

A more realistic situation is one in which the function produced by a function somehow exhibits consequences of the circumstances that pertained when, so to speak, it was brought into existence. Consider, for example,

```
LET G(X) BE
    LET F(Z) BE
        RETURN Z ↑ 2 + X
    END
    RETURN F
END
```

We would now expect

```
FA ← G(1)
FB ← G(2)
```

to produce two different functions, for we would expect FA(3) to yield 10 and FB(3) to yield 11.

We have crossed a crucial threshold here in that we permitted the function F to have a free variable. Observe that value X is certainly

bound in G, but that G is an outer block with respect to the block corresponding to F. Within the F block no binding of X can be found. To bring strong light to bear on this fact, we can recreate the example we've already treated in detail just above.

Recall that at one stage of the execution of the program shown above the symbol table was

S_o

A	2
H	$(\lambda\ X.\ \ldots)$
C	$(\lambda\ Z.\ Z \uparrow 2)$
D	ω

Had we defined F(Z) to be $(\lambda\ Z.\ Z \uparrow 2 + X)$ instead of $(\lambda\ Z.\ Z \uparrow 2)$ the value of C would have been

$$(\lambda\ Z.\ Z \uparrow 2 + X)$$

When now C is applied to 3 (as dictated by the step "D ← C(3)") the environment becomes

S_o

A	2
H	$(\lambda\ X.\ \ldots)$
C	$(\lambda\ Z.\ Z \uparrow 2 + X)$
D	ω

S_1

S_o	
Z	3

The interpreter then has the task of evaluating the expression

```
Z ↑ 2 + X
```

in the context provided by that environment. But that is impossible for no X can be found in it! Had an X been present in the table (suppose the first statement of the entire program had been "X ← 100"), it would surely not have been the one intended in the present context.

To make the situation quite clear, let's analyze a complete program.

```
DEFINE G(X) TO BE
   LET F(Z) BE
      RETURN  Z ↑ 2 + X
   END
   RETURN F
END
FA ← G(1)
FB ← G(2)
TYPE FA(3), FB(3)
EXECUTE
```

We would, as stated earlier, expect the program to produce the output

```
FA(3) = 10
FB(3) = 11
```

The job of the function G is to deliver the function (λ Z. Z ↑ 2 + X) as its value. The variable F plays no role whatever in G other than being a handle on what G is to ultimately deliver. It is, in other words, pure syntactic sugar. We can write G somewhat more clearly and compactly as

```
DEFINE G(X) TO BE
   RETURN (λ Z. Z ↑ 2 + X)
END
```

(An even more compact way to write it, by the way, would be "G ← (λ X. λ Z. Z ↑ 2 + X)".) When the G(1) is called, G will have to deliver (λ Z. Z ↑ 2 + X) but with the additional information that, in this instance, X is bound to 1. When G(2) is called subsequently, G must again deliver (λ Z. Z ↑ 2 + X) but then X is to be bound to 2. The two functions are therefore not the same.

It might occur to the reader that a good way of dealing with the problem here raised is to replace all the free variables in the function to be delivered by their values just before delivery. That solution would impose a considerable bookkeeping burden on the system. Apart from that, it fails to work in all cases. Consider, for example, the following slight modification of the above program.

```
DEFINE F(X) TO BE
RETURN (λ Z. Z ↑ 2 + X + A)
END
FA ← G(1)
A ← 10
etc.
```

Now both X and A are free within the function produced by G but A is not given a value until after G has been called. What is required therefore, is that a function be made to remember where, i.e. in what environment, values of its free variables are to be found when the function is applied -- not what their values were at the time the function is constructed.

The environment structure we have so far described consists of a number of nodes each of which, except for the topmost node, has a pointer to the just previous node. Each of these pointers serves the dual functions of

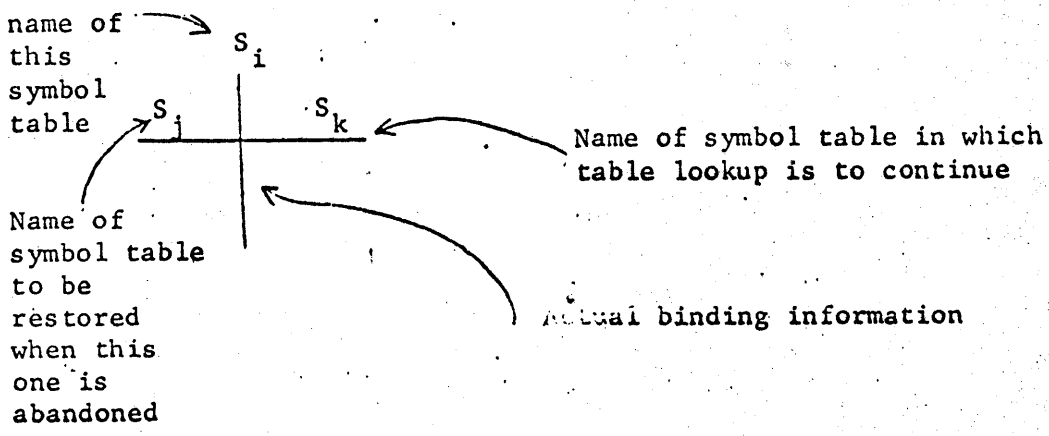
1) indicating where the search for a particular variable is to be continued if it hasn't yet been located in the structure so far interrogated

and

2) specifying what environment is to be restored upon the completion of the application of some function.

We now see that these two functions must be separated. We therefore introduce two pointers, the first (restoration pointer) pointing to the environment to be restored at the appropriate time, and the second (search pointer) indicating the search path to be pursued.

We then represent a typical symbol table as follows:



We must now add to each function a pointer to the environment in which it is to be evaluated. We call such a pointer an environment knot, or simply knot, and write a λ expression with its associated knot as in the following example:

$$(\lambda\ X, Y. \ldots)::S_m$$

This may be read as "the function of x and y that is such and such and is to be applied (to some arguments) in the environment S_m".

The idea that needs to be understood at this point is that of the evaluation of a λ expression. It is clear that, when the assignment statement

Y ← A + B

is executed, i.e. when the replacement operator " ← " is applied to (Y, A + B), both its operands must be evaluated. The evaluation of Y yields a location (sometimes called the "left hand value" of Y) and that of A + B a sum, i.e. presumably a number. What then should the evaluation of the λ expression that is the right hand side of the assignment statement

F ← (λ Z. Z ↑ 2 + X)

yield? More generally, what do we mean by the value of such a λ expression? We mean by it a λ expression (presumably a copy of the given one) knotted to its environment; i.e. to the environment in which the values of its free variables are to be found. A λ expression not knotted to any environment is called an open λ and one that is knotted a closure (following Landin). The result of evaluating an open λ is thus a closure.

Let's now simulate the execution of the program shown above.

Step 1

DEFINE G(X) TO BE

RETURN (λ Z. Z ↑ ? + X)

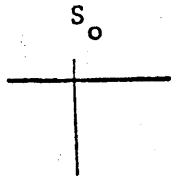
END

This step is equivalent to

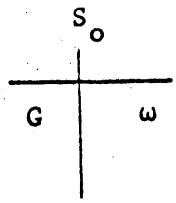
G ← (λ X. λ Z. Z ↑ 2 + X)

The symbol table grows as follows:

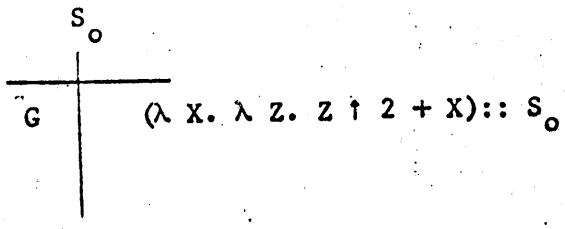
i)



ii)



iii)



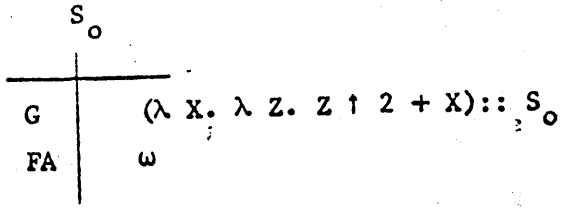
Note that the value of G is a closure. Its knot is to the environment that was regnant at the time the closure was formed, in this case the only environment around.

Step 2

$$FA \leftarrow G(1)$$

After the statement is recognized as an assignment statement

i)



A new node is created because a function is about to be applied.

ii)

S_o

G	$(\lambda\ X.\ \lambda\ Z.\ Z \uparrow 2 + X)::\ S_o$
FA	ω

S_1

S_o	S_o
X	1

Note that S_o is both the environment to be restored when the function application is completed and that into which table lookup is to propogate.

The open λ "$(\lambda\ Z.\ Z \uparrow 2 + X)$" is now evaluated and the resulting closure assigned to FA.

iii)

S_o

G	$(\lambda\ X.\ \lambda\ Z.\ Z \uparrow 2 + X)::\ S_o$
FA	$(\lambda\ Z.\ Z \uparrow 2 + X)::\ S_1$

Step 3

$$FB \leftarrow G(2)$$

The development is essentially the same as in the previous step.

i)

S_o

G	$(\lambda\ X.\ \lambda\ Z.\ Z \uparrow 2 + X)::\ S_o$
FA	$(\lambda\ Z.\ Z \uparrow 2 + X)::\ S_1$
FB	ω

ii)

S_o

G	
FA	as above
FB	

S_2

S_o	S_o
X	2

iii)

S_o

G	$(\lambda\ X.\ \lambda\ Z.\ Z \uparrow 2 + X)::\ S_o$
FA	$(\lambda\ Z.\ Z \uparrow 2 + X)::\ S_1$
FB	$(\lambda\ Z.\ Z \uparrow 2 + X)::\ S_2$

Step 4

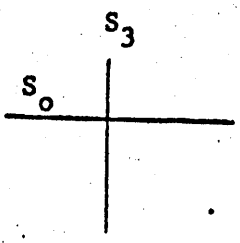
TYPE FA(3), FB(3)

First FA(3) is called. The closure

$(\lambda\ Z.\ Z \uparrow 2 + X)::\ S_1$

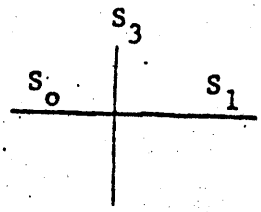
is found to be the value of FA. Because a function is to be applied, a new symbol table is created and tied to the current RST by means of the first pointer.

i)



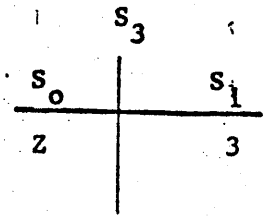
The knot on the closure provides the pointer to the search continuation environment

ii)



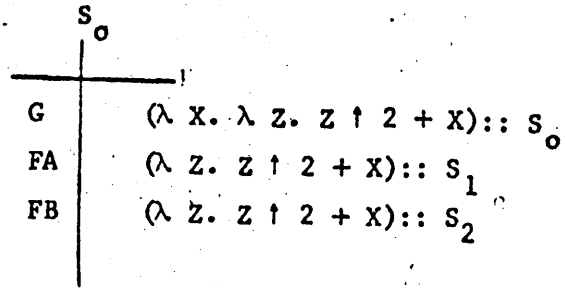
and the bound variable is placed as usual

iii)



The entire environment is now

iv)



S_1

S_o	S_o
X	1

S_3

S_o	S_1
Z	3

It is now easy to see that the evaluation of

$$Z \uparrow 2 + X$$

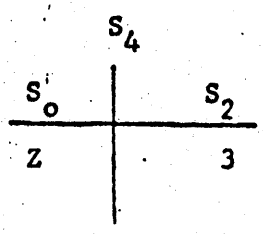
i.e. the body of FA, will yield 10. We assume that value is typed by the machine. The application of FB to 3 now proceeds just as above except that when the body of FB, which is the same as that of FA, is evaluated, the environment is

v) S_o

G	$(\lambda X.\ \lambda Z.\ Z \uparrow 2 + X) :: S_o$
FA	$(\lambda Z.\ Z \uparrow 2 + X) :: S_1$
FB	$(\lambda Z.\ Z \uparrow 2 + X) :: S_2$

S_2

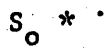
S_o	S_o
X	2



and, of course, FB(3) yields 11.

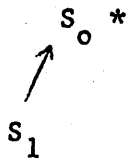
We have seen that the introduction of free variables in functions that are produced as values of functions required us to abandon a simple stack structured environment in favor of a tree structured environment. To imbed that idea firmly, consider the growth of the environment in the simulation just analyzed. Sketched in starkest terms it was:

i

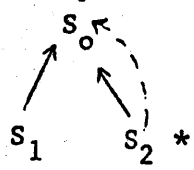


ii

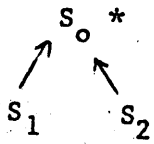
iii



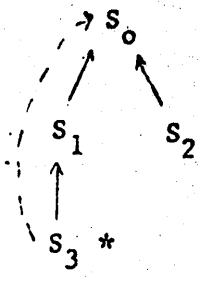
iv



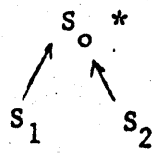
v



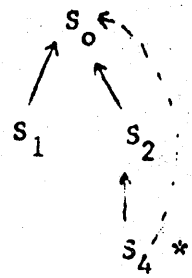
vi



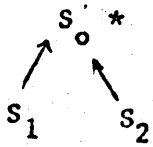
vii



vii



ix



Where, in the above picture

* indicates the regnant symbol table,

a solid arrow is the table look up path,

and

a broken arrow is the environment restoration path.

Notice that in step ii the symbol table S_1 appeared as the RST. That's when G was being applied to 1. In step iii S_o is again the RST but S_1 has not disappeared. The vitally important point here is that S_1 continues to be known to the system by virtue of the appearance of a pointer to S_1 as part of the closure

$$(\lambda\ Z.\ Z \uparrow 2 + X)::\ S_1$$

On the other hand, in step vi S_3 is the RST -- there FA(3) is being computed -- but disappears again in the next step vii. This is because no pointers to S_3 survive the restoration of the environment S_o as the RST. Analogous arguments apply to S_2 and S_4 respectively.

The general rule operative here is that <u>any structure survives as long as and only as long as a pointer to it exists anywhere in the system</u>. There is assumed to be a permanent variable <u>CURRENTE</u> that has a pointer to the RST as its value. Since every symbol table eventually points back to S_o, even if by a long chain of indirection, S_o is permanently safe. The appearance of an open λ or of a closure on either the intermediate result list IRL or on any symbol table constitutes a pointing to that open λ or closure. And, of course, the knot of any closure is always a pointer to some symbol table, keeping that symbol table and all its ancestors safe as long as the closure itself survives.

We have implicitly introduced a distinction between what are ordinarily called operators, e.g. the arithmetic operators + and ↑, and functions.

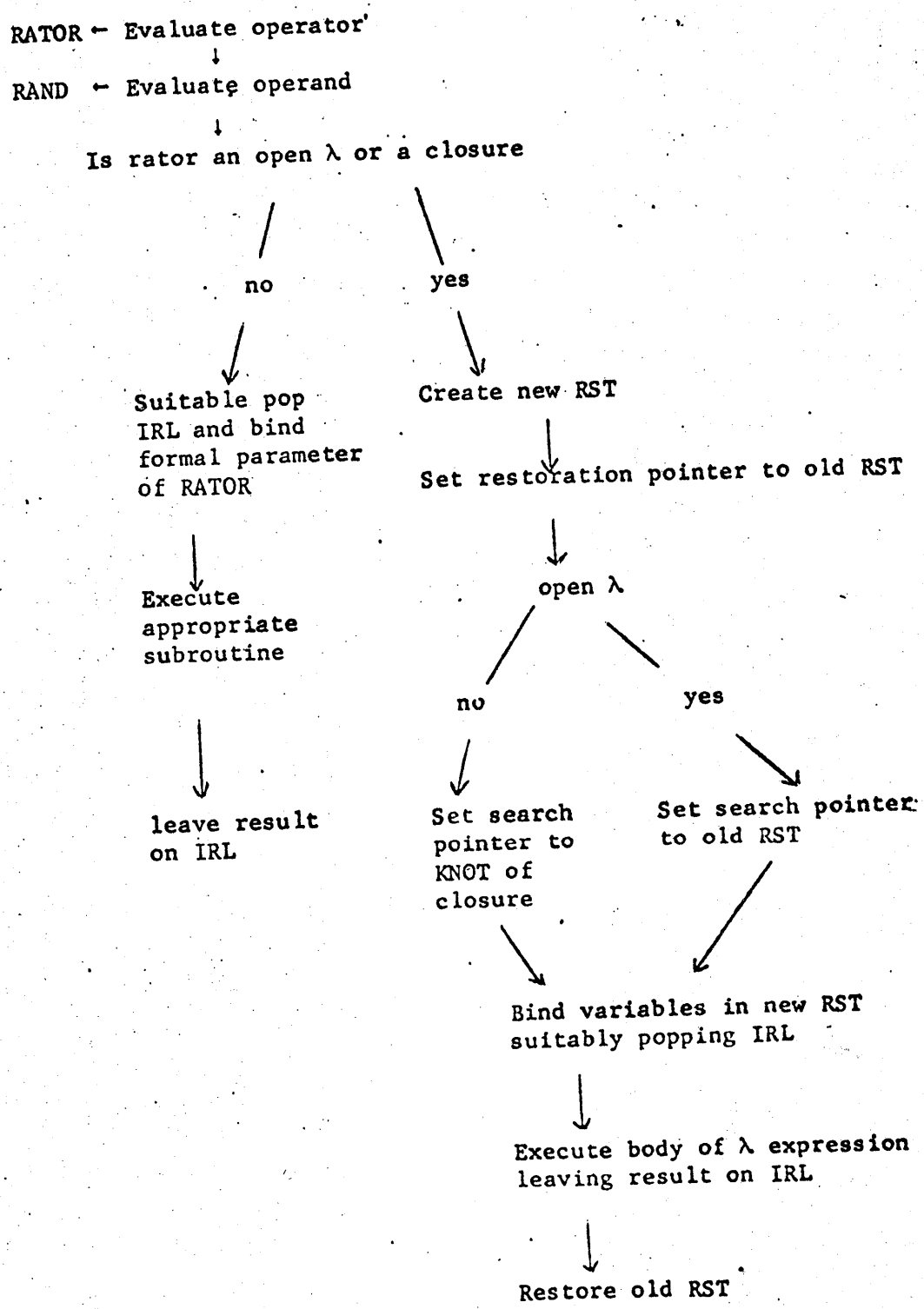
We should recognize, however, that we write "A+B" in preference to "+(A,B only for historical reasons. It is, in fact, useful to think of the "F" in the form "F(X)" as being fundamentally no different from the "+" in th form "+(A,B)", i.e. to think of both "+" and "F" as being simply two instances of operators. We hypothesize that our system has only one func tion, the function APPLY, and that it is purely internal to the system. We may imagine, for example, that the expression "F(A+B)" written by the programmer is, before evaluation, translated to the internal form "APPLY (F,(APPLY (+,(A,B))". APPLY is thus a function that always takes two arguments, an operator and an operand.

The evaluation of such an expression then proceeds as follows:

1) Evaluate operator part and call the result rator (again following Landin).
2) Evaluate operand part and call the result rand. (The rand is generally a set of values left on the IRL.)
3) Bind the formal parameters of the rator to the values given by the rand.
4) Execute the body of the program of the rator.

If the rator is a built in operator, such as one of the arithmetic operat then execution of its body means essentially going to the subroutine to which the rator points. If, however, the rator is a closure, then a new symbol table is created, its restoration pointer set to point to the then existing RST and its search pointer set to point to the symbol table knotted to the closure. The formal parameter binding then takes place in this new RST. If the rator is an open λ, then the procedure is as just stated except that the search pointer is also set to point to the old RST. Application of the rator to the rand then consists of executing the progr

that is the body of either the open λ or the closure. The result if left on the IRL. If a new RST was created, i.e. if the rator was not a built in operator, the old RST is, of course, restored upon completion of the application. This is shown in the flow diagram below.


RATOR ← Evaluate operator
RAND ← Evaluate operand
Is rator an open λ or a closure
no
yes
Suitable pop IRL and bind formal parameter of RATOR
Execute appropriate subroutine
leave result on IRL
Create new RST
Set restoration pointer to old RST
open λ
no
yes
Set search pointer to KNOT of closure
Set search pointer to old RST
Bind variables in new RST suitably popping IRL
Execute body of λ expression leaving result on IRL
Restore old RST

With the above in mind we can now work through an example involving the recursive operator factorial. Recursive operators are important in our context because they always involve the free occurrence in their bodies of the name of an operator.

Suppose then that the factorial function is produced as the value of a function. The function defined by

```
DEFINE H(X) TO BE
    IF X < 0 THEN
        RETURN (λ Z. Z + 1)
    ELSE
        LET F ← (λ N. IF N = 0 THEN 1 ELSE N*F (N-1))
        RETURN F
    END
```

will produce the successor function if given a negative argument, otherwise the factorial function.

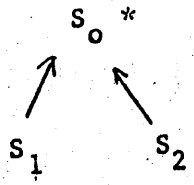
Suppose now that that function H had been defined as shown and we now come to a section of program

```
    .
    .
    .
F ← H(-1)
P ← H(0)
X ← P(F(1))
    etc.
```

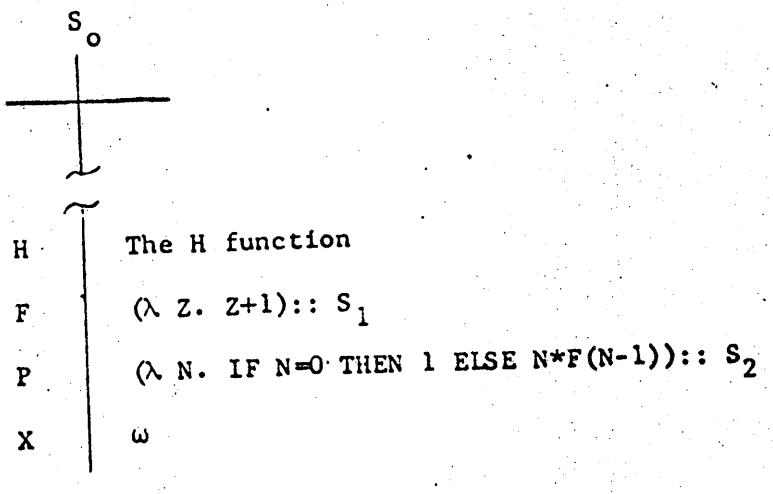
where the RST is S_o and, apart from H, has nothing of interest to show us. One might think the programmer is being foolish in using "F" for an identifier in his situation since H defines another function F. But the whole idea is that the behavior of H should in no way hinge on the names chosen for its local variables.

We may now follow the growth and shrinkage of the environment in a somewhat abbreviated representation. Again a "*" will mark the RST, a solid arrow the table look up path, and a broken arrow the environment restoration path. In addition, we show any particular symbol table in detail whenever its content changes.

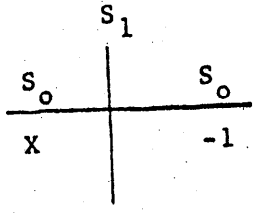
Nothing very interesting happens until after "X ← F(F(1))" has been reorganized as an assignment statement with X as the subject variable. The symbol table situation then is



where S_o is



and S_1



and S_2

S_2

S_o	S_o
X	0
F	(λ N. IF N=0 THEN 1 ELSE N*F(N-1)):: S_2

We then come to the evaluation of

P(F(1))

Recall that the internal representation of that is

APPLY (P, APPLY (F,1))

evaluation of the rator (i.e. P) yields

(λ N. IF N=0 THEN 1 ELSE N*F(N-1)):: S_2

We may think of that as being pushed unto a rator stack. Evaluation of the rand is, of course, evaluation of

APPLY (F, 1)

Evaluation of the rator of <u>that</u> yields the closure

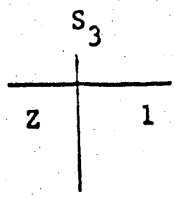
(λ Z. Z+1):: S_1

and of its rand 1. The value of the original rand is thus the result of applying (λ Z. Z+1):: S_1 to 1. Because the rator is a closure we construct a new symbol table and tie it up as perscribed above

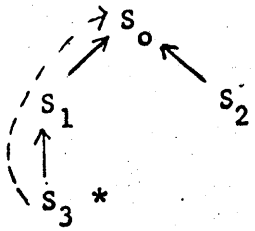
S_3

S_o	S_1

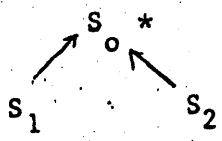
and bind the variable Z in it



The environment in which Z+1 is then evaluated is



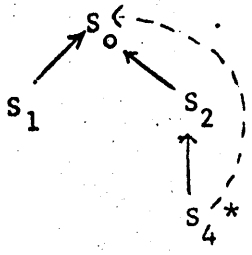
After Z+1 has been computed and 2 left on IRL the environment is again



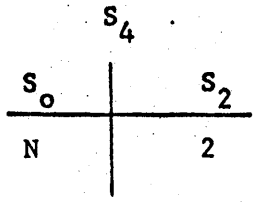
The rand of APPLY (P, APPLY (F,1)) has now been evaluated and the system set to apply the rator

$$(\lambda\ N.\ \text{IF } N=0 \text{ THEN } 1 \text{ ELSE } N*F(N-1))::\ S_2$$

to 2. A new symbol table is formed according to the rules stated above so that the new environment becomes



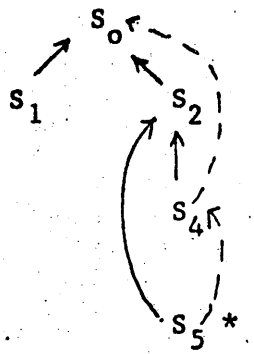
where S_4 is

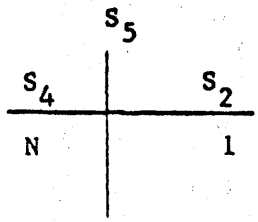


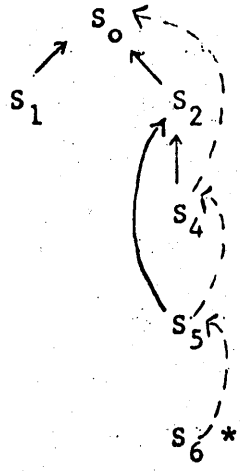
The body of the relevant closure dictates that F(N-1) is to be evaluated next so that later the result of that evaluation may be multiplied by N. Notice that F is a free variable in the body of the λ expression here under consideration. There is no problem in evaluating the rand of APPLY (F,(N-1)) for the value of N is to be found in the current RST, namely S_4. But there are now two F's in the system. The search pointer leads to S_2 where the value of F is found to be the closure

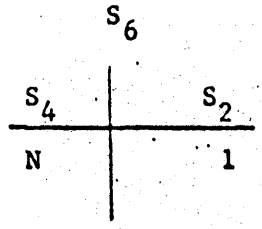
$$(\lambda\ N.\ \text{IF } N=0 \text{ THEN } 1 \text{ ELSE } N*F(N-1))::\ S_2$$

Following the rules of environment structuring repeatedly, will then result in the environment history shown below.

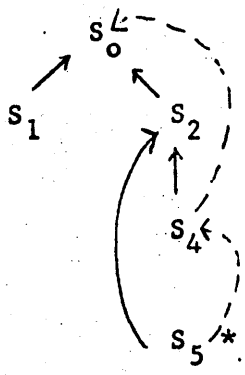


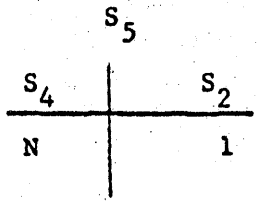




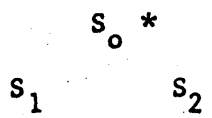


At this stage N=0 is left on the IRL. One restoration process is carried out and the environment returns to





so that when the computation N*F(N-1) is carried out the correct value of N, i.e. 1 in this case, is found in the environment. The process continues until, obviously, the environment state



is again reached and 2 left on IRL. Finally the assignment to X is made in S_o.

The reader may test his understanding of the entire mechanism by hand simulating the following program

```
LET TWICE (F) BE
    RETURN (λ X. F(F(X)))
END
LET THRICE (F) BE
    RETURN (λ Z. F(F(F(X))))
END
LET SUCCESSOR (N) BE
    RETURN N+1
END
TWTH ← TWICE (THRICE)
THTW ← THRICE (TWICE)
TWICE ← 0
THRICE ← 0
S9 ← TWTH (SUCCESSOR)
S8 ← THTW (SUCCESSOR)
SUCCESSOR ← 0
A ← S8(0)
B ← S9(0)
```

A should finally have 8 as its value and B 9. The reader should also satisfy himself that, given the above definitions, TWICE (THRICE (SUC)) is a different function than S9 and that TWICE (THRICE (SUC(0))) doesn't make any sense at all.

We made the point earlier that a variable that has a function, or more precisely an open λ or a closure, as its value behaves no differently from

any other variable. That remains true. When we, remembering that fact, note that in any ordinary arithmetic expression any variable may always be replaced by its value we are led to believe that that must also hold for function designating variables. If, for example, the value of A is 3, we would not change the value of the arithmetic expression

A + F(A)

be rewriting it

3 + F(3)

Similarly, if the value of F were the open λ (λ X. X+1), we would replace A + F(A) by A + (λ X. X+1) (A). The expression

(TWICE (SUCCESSOR)) (0)

(again appealing to the definitions developed above) could, to give still another example, be written

((λ F. λ X. F(F(X))) (λ N. N+1)) (0).

Such substituions are indeed permitted in our system.

The main practical significance of the fact that all variables are treated alike is that a system built to incorporate that principle is simpler and cleaner, i.e. more nearly free of ad hoc mechanisms, than one that distinguishes among several classes of variables. However, the theoretical implications of a language so constructed are far reaching. In the main, they lead to the possibility of arriving at a canonical language in terms of which many languages may be compared and their semantics clarified. Landin in paritulcar has pushed this idea very hard and has succeeded in analyzing ALGOL 60 in such terms. (See Landin - 1965.)

While it may not be obvious, it is nevertheless true, that any algorithm that can be expressed at all can be expressed in the form of the kind of operator-operand pairs shown above where either or both members of the pair may be open λ's. One may, in other words, write any program as a

(possibly large) nest of open λ's and constants. But there is a difficulty with conditional expressions. Suppose, for example, that we wish to express

IF A = 0 THEN 1 ELSE 1/A

in the form of λ expressions.

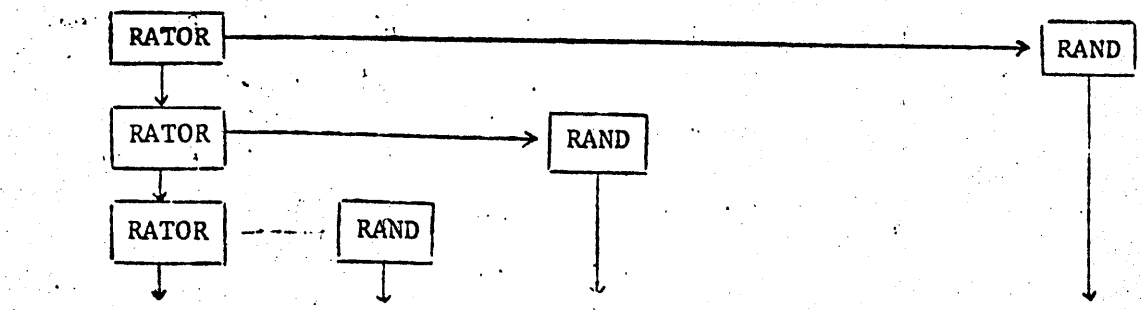
Let's assume we have a built in operator "IF" with the following characteristics:

1) IF operates on a boolean expression
2) If the value of the operand is TRUE, then the value of the IF expression is the operator FIRST, otherwise its value is the operator SECOND
3) The operator FIRST has as its value the value of the first expression of the two that are its arguments
4) The operator SECOND has as its value the value of the second expression of the two that are its arguments.

Then

((IF(A=0)) ((λ X. 1), (λ z. 1/A))) (A)

has the expected value. In the picture shown we decompose this expression into its rator-rand components and their values for the cases A=0 and A≠0



	IF	(A=0)	((λ X. 1), (λ X. 1/A))	(A)
A=0	FIRST	TRUE	(λ. 1)::S_i, (λ X. 1/A)::S_i	0
A≠0	SECOND	FALSE	(λ. 1)::S_i, (λ X. 1/A)::S_i	non-zero

The point is that all the tip nodes of the rator-rand tree could have been evaluated simultaneously. The evaluation of the rand

$$((\lambda\ Z.\ 1),\ (\lambda\ Z.\ 1/A))$$

produced the two closures

$$(\lambda\ X.\ 1)::\ S_i$$

and

$$(\lambda\ X.\ 1/A)::\ S_i$$

but did not result in an evaluation of either, hence avoiding the evaluation of 1/A for A=0. The value of

$$(IF\ (A=0))$$

then selects the appropriate closure which is then applied to the final argument A.

The reason we needed all this elaborate trickery is that we assumed that all rators and rands of a given expression are always evaluated -- inde that they may be evaluated simultaneously, once the expression is suitably decomposed. One consequence of this assumption is that we cannot look at a function, so to speak, and ask whether or not it wants its arguments evaluat at all, or evaluated in some special way.

Suppose then that we impose an order of evaluation, namely one that requires that rators be evaluated before rands. We could then design at least built in functions such that they take unevaluated rands and do whatever is appropriate with them under the circumstances. The subroutine associated with IF, for example, could be constructed to assume three argum

$$IF\ (A_1,\ A_2,\ A_3)$$

which are handed to it unevaluated. The first is then subjected to evaluation under control of the IF program itself and subsequently either A_2 or A_3 evaluated depending on the outcome of the evaluation of A_1.

The pressure to impose an order of evaluation doesn't really come from considerations such as those just discussed. For difficulties arising out of conditional expressions can usually be resolved by expression pre-processors whose main functions are syntactic checking and syntactic sugar removal. The assignment operator imposes a real difficulty. So much so, that a system permitting assignment must necessarily lose some purity. This is because the assignment operator requires that the first of its two operands be evaluated in a special way, i.e. for a left hand value (loosely speaking an address). This means that that operator, hence all operators, must be inspected before argument evaluation can proceed. But there is another operator we would dearly like to have. This is the QUOTE operator. We shall write it as a single apostrophe. Its function is simply to prevent the evaluation of the expression it quotes.

Recall that the assignment statement

$$F \leftarrow (\lambda\ X.\ ...)$$

causes a closure to be assigned to F, not an open λ. The effect of that is two fold. One that the environment that is regnant at the time of assignment is automatically secured against erasure, and the other that the application of F to its arguments is carried out in the context of that environment. But we may wish to assign a function to F which when later invoked will operate in the context of that subsequent invokation. In order to achieve that we will have to assign an open λ to F. The QUOTE operator permits this for with it we may simply write

$$F \leftarrow '(\lambda\ X.\ ...).$$

But now suppose F had been assigned an open λ and we then wish to assign the closure of the value of F to G. Were we to execute

$$G \leftarrow F$$

the open λ would be communicated to G since, of course, the evaluation of F yields only that open λ. What we need is an operator to force an evaluation. Such an operator is usually called EVAL.

G ← EVAL(F)

has the effect described above. EVAL can thus seem to be an anti-quote operator. It follows, for example, from arguments already stated that we can write

G ← EVAL('(λ X. ...))

for the above without changing its effect. What we have done by first applying a QUOTE and later EVAL is to postpone evaluation. The utility of that is that we may construct a new environment in the meanwhile and that it is that new environment that determines the values of the free variables appearing in the body of the λ expression.

We finally introduce a slight generalization with the operators CEVAL and CLOSE. Obviously the value of EVAL is the value of the expression being evaluated. Thus, the value of

EVAL ('(1+2))

is 3. CEVAL, on the other hand, leaves as its value the environment created during the course of evaluating the expression (i.e. program) that is given it as an argument. To give the simplest example, consider

K ← CEVAL ('(LET A ← 1, LET B ← 2))

K becomes a pointer to a symbol table

S

A	1
B	2

The operator CLOSE takes as one of its arguments an open λ and as its other such a symbol table pointer and leaves as its value the closure consisting of that open λ knotted to the symbol table pointed to.

If then F has an open λ as its value, the statement

(CLOSE (F, CEVAL('( )))) ( )

given here is skeletal form will, when executed, cause F to be applied to its arguments (enclosed in the last parenthesis pair) in a context provided by the environment determined by the quoted program that is the argument to CEVAL. If a given statement has a facility for reading environments from a secondary store, say a READE operator reads a disk file specified by its arguments into core and has a pointer to the read file as its value, then the statement

(CLOSE (F, READE ( ))) ( )

would cause F to be applied in the context provided by such a read in environment. This would allow the experimental evaluations of functions against prestored environments.

Any actual implementation of an interpreter incorporating the mechanisms here described must offer not only a guarantee that programs written for it have the expected outcome, i.e. that the identifier collision problem is really solved in all cases, but that the dynamic space allocation problem that must inevitably arise with the creation and disappearance of elaborate symbol table substructures also be solved. We touched on that problem when we said earlier that "any structure survives as long as and only as long as a pointer to it exists anywhere in the system". Let us now explore this issue a little more deeply.

We consider a fairly complex program in detail, this time with special attention to the appearance and disappearance of data structures.

TW ← (λ F. λ X. F(F(X)))

TH ← (λ F. λ X. F(F(F(X))))

TWTH ← TW(TH)

DELETE TW, TH

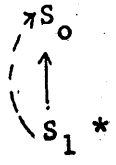
S ← (λ N. N+1)

P ← (TWTH(S)

DELETE S, TWTH

The DELETE command causes the variables given it as arguments to be removed from the environment in force at the time of its encounter.

After the first two assignments have been made and the third statement recognized as an assignment statement and the execution of TW(TH) begun, we have



where S_o is

S_o

TW	(λ F. λ X. F(F(X))):: S_o
TH	(λ F. λ X. F(F(F(X)))):: S_1
TWTH	ω

and S_1

S_1	
S_o	S_o
F	$(\lambda\ F.\ \lambda\ X.\ F(F(F(X))))::\ S_o$

Note that TW(TH) calls for the application of

$$(\lambda\ F.\ \lambda\ X.\ F(F(X)))$$

to

$$(\lambda\ F.\ \lambda\ X.\ F(F(F(X))))::\ S_o$$

In S_1, F has been bound to the closure shown just above. We then come to the evaluation of the body of the rator. That body is

$$(\lambda\ X.\ F(F(X)))$$

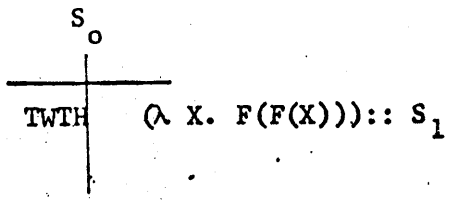
Its evaluation in the RST results in the closure

$$(\lambda\ X.\ F(F(X)))::\ S_1$$

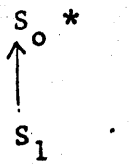
and when that is left on IRL and ultimately assigned as the value of TWTH in S_o, the symbol table S_1 ceases to be regnant and is, in a sense, abandoned. Hence the environment after the third assignment of our program is simply S_o, i.e.

S_o	
TW	$(\lambda\ F.\ \lambda\ X.\ F(F(X)))::\ S_o$
TH	$(\lambda\ F.\ \lambda\ X.\ F(F(F(X))))::\ S_o$
TWTH	$(\lambda\ X.\ F(F(X)))::\ S_1$

and after deletion of TW and TH



But a pointer to S_1 has survived -- namely on the closure that is the value of TWTH. Hence S_1 itself survives. The RP of S_1 was removed when the symbol table to which it pointed (S_o) was restored to regnant status. But the SP of S_1 (in this case also pointing to S_o) is a permanent part of S_1. We are thus entitled to represent the current environment as



i.e. to consider S_1 and its SP as still being part of the game.

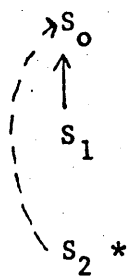
The execution of the statement

$$S \leftarrow (\lambda N.\ N+1)$$

has the sole effect of augmenting S_o with the identifier value pair

$$S \mid (\lambda N.\ N+1)::\ S_o$$

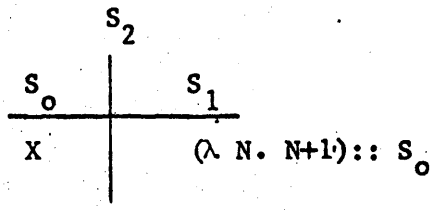
By the time we come to evaluate TWTH(S) we have had to create a new symbol table S_2. We thus have



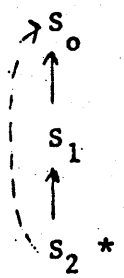
we could set the RP of S_2 immediately because it is always set to the environment that was regnant when the new symbol table was established. To set the SP, we must look at the closure of the function we are about to invoke. In this case it is

$$(\lambda\ X.\ F(F(X)))::\ S_1$$

We therefore construct S_2 as follows:



(We found the value to be assigned to X in S_o, of course.) The environment now is



Having tied up the new symbol table appropriately and bound the formal variable of the function, we are supposed to apply, we must now execute the program that is the body of the closure. We must, in the present example evaluate

$$F(F(X))$$

Recalling that what we are really evaluating is

APPLY (F, APPLY (F X))

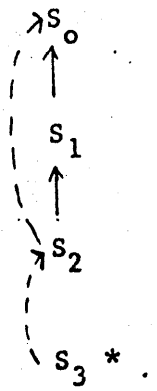
we know that we first evaluate the inner F(X). This calls for the creation of a new symbol table S_3 with RP of S_2. We search for F starting in S_2 and find the closure

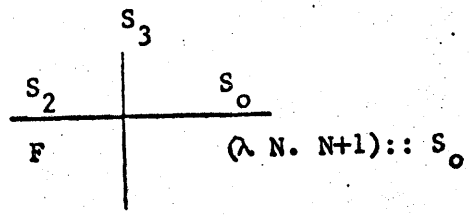
$$(\lambda\ F.\ \lambda\ X.\ F(F(F(X))))::\ S_o$$

in S_1. Its formal parameter is F and its knot S_o. We find the value of X (remember we are doing F(X)) in S_2 to be the closure

$$(\lambda\ N.\ N+1)::\ S_o$$

and now have sufficient information to construct and attach S_3 properly.





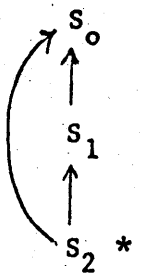
The body of the closure we are currently applying is

$$(\lambda\ X.\ F(F(F(X))))$$

and its value in the current context is the closure

$$(\lambda\ X.\ F(F(F(X))))::\ S_3$$

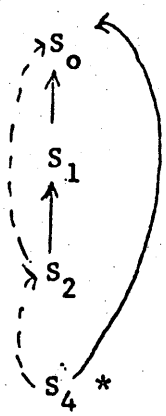
Once we have that in IRL we may abandon S_3 and restore the old RST accordin to the RP of S_3. We thus have

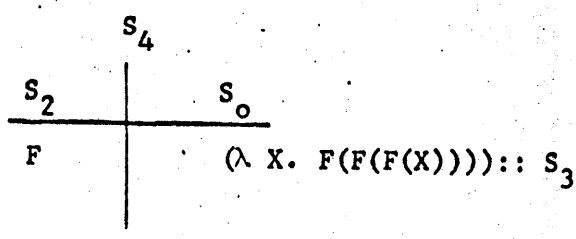


But S_3 survives because a closure bearing a pointer to it is on IRL. When we now come to apply the outer F of the expression

APPLY (F, APPLY (F X))

We again find the F in S_1 and essentially repeat what we have just done except that we deal with S_4 in place of S_3. For a moment we have

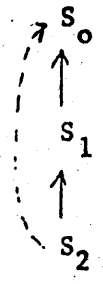




and after leaving the closure

(λ X. F(F(F(X)))):: S_4

on IRL

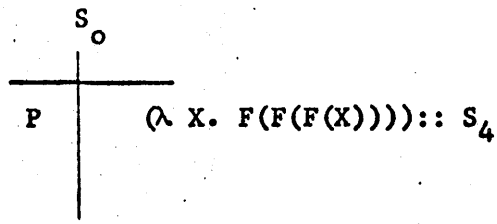


where again S_4 survives because a closure pointing to it is on IRL. When we finish the second application of F, however, we are also finished with the body of the function we were applying. We must therefore follow the RP of the currently regnant symbol table S_2 and restore S_o to regnant status and there make the appropriate assignment. When, thus, we come to the end of the execution of the steps

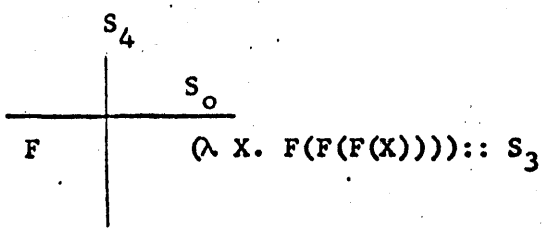
P ← TWTH(S)

DELETE S, TWTH

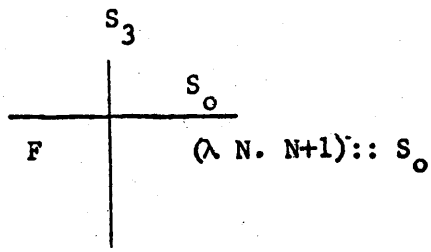
we have the environment



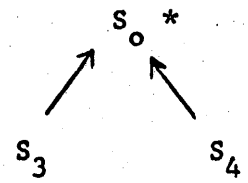
That contains a pointer to S_4



and that a pointer to S_3



We see then that the structure that has survived is



and that is the minimum structure required to, for example, compute P(1). The reader should test his understanding of what was said here by simulating that computation.

We have shown a way of handling variables and their values in a way that permits functions to have functions as their values, i.e. to deliver functions to higher levels of activation. We have seen that the main problems that must be solved in this connection is that of preventing identifier collisions especially in the case in which delivered functions have free variables. Our solution to these problems is mainly that of providing a tree structured symbol table in place of the more usual stack structured symbol table. That solution is, of course, not original with us. It shows up, for example, in some versions of LISP and in Landin's SECD machine. Landin, however, insists that no order of evaluation be imposed on rator-rand pairs and thus excludes the QUOTE operator. We have shown some uses of that operator -- in particular, that it permits open λ expressions to be passed around freely to be closed later by association with arbitrary and perhaps experimental environments.

REFERENCES

Bobrow, D. G. and Murphy, D. C. (1967) Structure of a LISP system using two level storage, Communications of the A.C.M., 10, 155-159. (See especially the footnote on p. 158.)

Landin, P. J. (1965) A correspondence between ALGOL 60 and Church's λ notation, Communications of the A.C.M., 8, 89-101, 158-65.

Landin, P. J. (1966) A λ calculus approach, in Advances in Programming and Non Numerical Computation, FOX, L. (ed.), Pergamon Press, New York, 1966.

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine, (Part 1), Communications of the A.C.M., 3, 184-95.

McCarthy, J., et al. (1962) LISP 1.5 programmer's manual. M.I.T.