A FUNCTIONAL LANGUAGE

The imperative languages we have studied so far all share one thing; they have assignable variables. We have seen several ways of handling the storage problem associated with variables, culminating in the stack-based model which is familiar on most modern languages. However, at the same time that FORTRAN and COBOL were getting their start, the functional language LISP was using another paradigm for expressing algorithms. Like all functional languages, LISP has no variables in the imperative sense. Variables in LISP are just like the lambda variables in our denotational language. In fact, LISP was developed as a programming language from work in computation theory using the lambda calculus, so it is not wonder that they share this feature. We will first look at a very simple pure functional language very much like basic lambda calculus. Since it operates very much like the denotational language, its semantics are very simple.

Programs in this language are just expressions. Since there are no variables there is no need for assignment, and thus no commands. Initially we will not include arithmetic, but instead have the language represent and operate on lists, just like the original LISP (the LISP Programming language). Lists are comprised of either atoms, an infinite set of primitive objects, or other lists. We immediately have a problem since this is a recursively defined domain. To make matters worse, functions will map such values into other such values. If the domain is called Denotable-value, then lists will be Denotable-value*, and functions will be Denotable-value → Denotable-value. We will be able to bind functions to identifiers, so functions are themselves members of Denotable-value. We thus have the domain:

\[
\text{Denotable-value} = (\text{Denotable-value} \rightarrow \text{Denotable-value}) + \text{Denotable-value}^* + \text{Atom} + \text{Errvalue}
\]

where the domain Atom is primitive, and Errvalue = Unit.

This is an equation very much like the recursive function equations studied earlier. We will not show the solution to this here, but treat it in a simple operational fashion.

THE ENVIRONMENT

If there are no variables in the language, there is no need for a store. In fact, we can operate solely with an environment, which will allow us to bind identifiers to denotable values, and also give us the context mechanisms to allow reuse of names. The environment will thus be

Domain Environment = Id → Denotable-value

accessenv : Id → Environment → Denotable-value
accessenv = \lambda i.\lambda e.(i)

updateenv : Id → Denotable-value → Environment → Environment
updateenv = \lambda i.\lambda d.\lambda e.[i \mapsto d]e

and we will define the Denotable-value domain as

Domain Denotable-value = (Function + List + Atom + Error)⊥

where Function = Denotable-value → Denotable-value,
List = Denotable-value*,
Error = Unit

To these we will add the list operations hd, tl, null and cons, with the empty list nil.

These are the only semantic algebras since there is no arithmetic.

SYNTAX

The syntax is almost trivial since there is only one real non-terminal, Expression.

E ∈ Expression
A ∈ Atomic-symbol
\[ \text{E} ::= \text{LET} \, I = \, \text{E} \mid \text{LAMBDA} \, (I) \, \text{E} \mid \text{CONS} \, \text{E}_2 \mid \text{HEAD} \, \text{E} \mid \text{TAIL} \, \text{E} \mid \text{NIL} \]

All of these forms are familiar from the lambda calculus itself. LET introduces a new name and binds it to the value of an expression. LAMBDA is the form for a function, with single parameter \( I \) and body \( \text{E} \). CONS, HEAD, TAIL, and NIL are the explicit forms of \texttt{cons}, \texttt{hd}, \texttt{tl}, and \texttt{nil} in the algebra. It is clear from the domain of denotable values that functions are first-class objects and can be bound to identifiers just like lists, which are data values.

The language will use static scoping rules, since we need to disambiguate references in the nested contexts of LET, as in this example:

\[
\begin{align*}
\text{LET} \, F = & \, a_0 \, \text{IN} \\
\text{LET} \, F = & \, \text{LAMBDA} \, (Z) \, F \, \text{CONS} \, Z \, \text{IN} \\
\text{LET} \, Z = & \, a_1 \, \text{IN} \\
F & \, (Z \, \text{CONS} \, \text{NIL})
\end{align*}
\]

Here the first \( F \) is bound to the atom \( a_0 \) and the \( F \) in the body of the function is this value. The function is also bound to \( F \), but the scoping rules do not make this a recursive function (we will do this later). The semantics will be shown to be equivalent to the substitution rules of the lambda calculus. Using these rules, expression simplifies to

\[
\begin{align*}
\text{LET} \, F = & \, a_0 \, \text{IN} \\
\text{LET} \, F = & \, \text{LAMBDA} \, (Z) \, a_0 \, \text{CONS} \, Z \, \text{IN} \\
\text{LET} \, Z = & \, a_1 \, \text{IN} \\
F & \, (Z \, \text{CONS} \, \text{NIL})
\end{align*}
\]

\[
\begin{align*}
= & \, \text{LET} \, F = \, \text{LAMBDA} \, (Z) \, a_0 \, \text{CONS} \, Z \, \text{IN} \\
= & \, \text{LET} \, Z = \, a_1 \, \text{IN} \\
F & \, (Z \, \text{CONS} \, \text{NIL})
\end{align*}
\]

\[
\begin{align*}
= & \, \text{LET} \, Z = \, a_1 \, \text{IN} \\
& \, (\text{LAMBDA} \, (Z) \, a_0 \, \text{CONS} \, Z) \, (Z \, \text{CONS} \, \text{NIL})
\end{align*}
\]

\[
\begin{align*}
= & \, (\text{LAMBDA} \, (Z) \, a_0 \, \text{CONS} \, Z) \, (a_1 \, \text{CONS} \, \text{NIL})
\end{align*}
\]

\[
\begin{align*}
= & \, a_0 \, \text{CONS} \, (a_1 \, \text{CONS} \, \text{NIL})
\end{align*}
\]

The valuation functions will ensure this semantics. Note that at each stage, once a binding has been done, the binding cannot change (there is no assignment), so functional programming can be called programming with constants.

**VALUATION FUNCTIONS**

There is only one function, the one for evaluating expressions. However there are ten alternatives:

The functionality of \( \text{E} \) is

\[
\text{E} : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Expressible} \rightarrow \text{value}
\]

where \( \text{Expressible} \rightarrow \text{value} = \text{Denotable} \rightarrow \text{value} \)

since these are the only values in the language. Note that functions can also be returned as the values of other functions. A function can even accept itself as an argument and return itself as a result.

The LET form binds its new identifier to the value of the expression:

\[
\text{E} \left[ \text{LET} \, I = \, \text{E}_1 \, \text{IN} \, \text{E}_2 \right] = \lambda \text{e.} \text{E} \left[ \text{E}_1 \right] \left( \text{updateenv} \left[ \text{I} \right] \right) \left( \text{E} \left[ \text{E}_1 \right] \right) \left( \text{e} \right)
\]

The environment is updated with the new binding and used to evaluate the expression in the body of the LET. A function is simply tagged as such and a lambda expression is formed from its parts.

\[
\text{E} \left[ \text{LAMBDA} \, (I) \, \text{E} \right] = \lambda \text{e.} \text{inFunction} \left( \lambda \text{d.} \text{E} \left[ \text{E} \right] \left( \text{updateenv} \left[ \text{I} \right] \right) \left( \text{d} \right) \left( \text{e} \right) \right)
\]
The function so defined can be executed when it is applied to some value \( d \).

An application needs to be careful that the first expression evaluates to a function, and the second expression does not evaluate to an error (some of the possible list operations can result in errors).

\[
E[E_1, E_2] = \lambda e. \text{cases } E[E_1] e \text{ of } \\
\begin{align*}
\text{isFunction}(f) & \rightarrow f(E[E_2] e) \\
\text{isList}(t) & \rightarrow \text{inError}() \\
\text{isAtom}(a) & \rightarrow \text{inError}() \\
\text{isError()} & \rightarrow \text{inError}() \\
\end{align*}
\]

The list operations simply translate into the corresponding algebraic list operation.

\[
E[E_1 \text{ CONS } E_2] = \lambda e. \text{let } x = E[E_2] e \text{ in } \\
\text{cases } x \text{ of } \\
\begin{align*}
\text{isFunction}(f) & \rightarrow \text{inError}() \\
\text{isList}(t) & \rightarrow \text{inList}((E[E_1] e) \text{ cons } t) \\
\text{isAtom}(a) & \rightarrow \text{inError}() \\
\text{inError()} & \rightarrow \text{inError}() \\
\end{align*}
\]

\[
E[\text{HEAD } E] = \lambda e. \text{let } x = E[E] e \text{ in } \\
\text{cases } x \text{ of } \\
\begin{align*}
\text{isFunction}(f) & \rightarrow \text{inError}() \\
\text{isList}(t) & \rightarrow (\text{null } t \rightarrow \text{nError}() \text{ cons } x) \\
\text{isAtom}(a) & \rightarrow \text{inError}() \\
\text{inError()} & \rightarrow \text{inError}() \\
\end{align*}
\]

\[
E[\text{TAIL } E] = \lambda e. \text{let } x = E[E] e \text{ in } \\
\text{cases } x \text{ of } \\
\begin{align*}
\text{isFunction}(f) & \rightarrow \text{inError}() \\
\text{isList}(t) & \rightarrow (\text{null } t \rightarrow \text{nError}() \text{ cons } \text{inList } (tl t)) \\
\text{isAtom}(a) & \rightarrow \text{inError}() \\
\text{inError()} & \rightarrow \text{inError}() \\
\end{align*}
\]

The remainder are the simple base cases:

\[
E[\text{NIL}] = \lambda e. \text{inList(nil)} \\
E[1] = \text{accessenv}[1] \\
E[A] = \lambda e. \text{inAtom}(A[A]) \\
E[(E)] = E[E] \\
\]

The \( A \) function simply returns the object corresponding to the identifier. E.g. \( A[a] = a, A[b] = b, \) etc.
AN EXAMPLE DERIVATION

We will work the example given above which was simplified according to lambda calculus rules. We will now use the proper valuation functions.

\[ E_0 = \text{LET } F = a_0 \text{ IN } E_1 \]
\[ E_1 = \text{LET } F = \text{LAMBDA } (Z) \text{ F CONS } Z \text{ IN } E_2 \]
\[ E_2 = \text{LET } Z = a_1 \text{ IN } F(Z \text{ CONS } \text{NIL}) \]

\[ E\left[ \text{LET } F = a_0 \text{ IN } E_i \right] e_0 = E\left[ E_1 \right] \left( \text{updateenv}[F] \left( E\left[ a_0 \right] e_0 \right) \right) \]
\[ = E\left[ E_1 \right] \left( \text{updateenv}[F] a_0 e_0 \right) \]
\[ = E\left[ \text{LET } F = \text{LAMBDA } (Z) \text{ F CONS } Z \text{ IN } E_2 \right] e_1 \]
\[ = E\left[ E_2 \right] \left( \text{updateenv}[F] \left( E\left[ \text{LAMBDA } (Z) \text{ F CONS } Z \right] e_1 \right) \right) \]
\[ = E\left[ E_2 \right] \left( \text{updateenv}[F] \left( \text{inFunction} \left( \lambda d. E\left[ F \text{ CONS } Z \right] \left( \text{updateenv}[Z] d e_1 \right) \right) \right) \right) e_1 \]
\[ = E\left[ E_2 \right] \left( \text{updateenv}[F] \left( \text{inFunction} \left( \lambda d. E\left[ F \text{ CONS } Z \right] e_f \right) \right) \right) e_1 \]
\[ = E\left[ \text{LET } Z = a_1 \text{ IN } F(Z \text{ CONS } \text{NIL}) \right] e_2 \]
\[ = E\left[ F(Z \text{ CONS } \text{NIL}) \right] \left( \text{updateenv}[Z] \left( E\left[ a_1 \right] e_2 \right) \right) \]
\[ = E\left[ F(Z \text{ CONS } \text{NIL}) \right] \left( \text{updateenv}[Z] a_1 e_2 \right) \]
\[ = E\left[ F(Z \text{ CONS } \text{NIL}) \right] e_3 \]

At this point the environment \( e_3 \) is

\[ \begin{bmatrix} Z \mapsto \text{inAtom}(a_1) \end{bmatrix} \begin{bmatrix} F \mapsto \text{inFunction} \left( \lambda d. E\left[ F \text{ CONS } Z \right] e_f \right) \end{bmatrix} \begin{bmatrix} F \mapsto \text{inAtom}(a_0) \end{bmatrix} e_0 \]

which is the same as

\[ \begin{bmatrix} Z \mapsto \text{inAtom}(a_1) \end{bmatrix} \begin{bmatrix} F \mapsto \text{inFunction} \left( \lambda d. E\left[ F \text{ CONS } Z \right] e_f \right) \end{bmatrix} e_0 \]

\( e_f \) is the update of \( \begin{bmatrix} F \mapsto \text{inAtom}(a_0) \end{bmatrix} e_0 \) used in the function. Note how the function already has its environment for execution; it was put in there at the time of binding it to \( F \), when the environment only included \( F \) bound to \( a_0 \). Continuing with the derivation:

\[ E\left[ F (Z \text{ CONS } \text{NIL}) \right] e_3 \]
\[ = \text{cases } E\left[ F \right] e_3 \text{ of } \]
\[ \begin{cases} 
\text{isFunction}(f) \rightarrow f \left( E\left[ Z \text{ CONS } \text{NIL} \right] e_3 \right) \\
\text{isList}(t) \rightarrow \text{inError}() \\
\text{isAtom}(a) \rightarrow \text{inError}() \\
\text{isError()} \rightarrow \text{inError}() 
\end{cases} \]
end
\[ = \left( \lambda d. E\left[ F \text{ CONS } Z \right] e_f \right) \left( E\left[ Z \text{ CONS } \text{NIL} \right] e_3 \right) \]

We need to simplify the argument first, using call-by-value semantics, so
\[ E[Z \text{ CONS} \text{ NIL}] e_3 \]
\[ = \text{let } x = E[NIL] e_3 \text{ in} \]
\[ \text{cases } x \text{ of} \]
\[ \quad \text{isFunction}(f) \rightarrow \text{inError()} \]
\[ \quad \text{isList}(t) \rightarrow \text{inList}(E[Z] e_3 \text{ cons } t) \]
\[ \quad \text{isAtom}(a) \rightarrow \text{inError()} \]
\[ \quad \text{inError()} \rightarrow \text{inError()} \]
\[ \text{end} \]
\[ = \text{inList}(E[Z] e_3 \text{ cons } \text{ nil}) \]
\[ = \text{inList}(\text{inAtom}(a) \text{ cons } \text{ nil}) \]

Applying the function, we get
\[ = (\lambda d. E[F \text{ CONS} Z] e_f)(\text{inList}(\text{inAtom}(a) \text{ cons } \text{ nil})) \]
\[ = E[F \text{ CONS} Z](\text{updateEnv}[Z]\text{inList}(\text{inAtom}(a) \text{ cons } \text{ nil}))e_f \]
\[ = E[F \text{ CONS} Z] e_f \]
\[ = \text{let } x = E[Z] e_f \text{ in} \]
\[ \text{cases } x \text{ of} \]
\[ \quad \text{isFunction}(f) \rightarrow \text{inError()} \]
\[ \quad \text{isList}(t) \rightarrow \text{inList}(E[F] e_f \text{ cons } t) \]
\[ \quad \text{isAtom}(a) \rightarrow \text{inError()} \]
\[ \quad \text{inError()} \rightarrow \text{inError()} \]
\[ \text{end} \]
\[ = \text{inList}(E[F] e_f \text{ cons } (\text{inAtom}(a) \text{ cons } \text{ nil})) \]
\[ = \text{inList}(\text{inAtom}(a_0) \text{ cons } (\text{inAtom}(a) \text{ cons } \text{ nil})) \]

which is the final result.