Programming Languages

Storing Variable Values

Scope

This article assumes you have read The Environment Model of Evaluation.

The term *scope* refers to the set of variables which are accessible at any execution point in a computer program. Most modern programming languages use *static* scope, meaning, one can resolve a reference to a non-local variable simply by looking at a printout of the code. If a variable is not defined in the local scope, successive outer scopes are searched until the variable is found.

One possible implementation of scope uses the *environment model.* An *environment* can be thought of as linked list of tables; each table (from front to back) holds the variables and their values for ever increasingly outer scopes. An environment can also be used during parsing to determine if a variable has been defined before use and to detect type errors in statically typed languages.

There are five environment routines that are part of an *environment passing* language interpreter:

- `insertEnv(e,v,w)` placing an variable *v* and its initial value *w* into the most local table of environment *e*
- `lookupEnv(e,v)` find the value of a variable *v* in environment *e*; tables are searched from most local to most outer; the value of the first occurrence of the variable is returned
- `updateEnv(e,v,w)` change the value of first occurrence of a variable *v* to *w* in environment *e*; tables are searched from most local to most outer
- `extendEnv(e,m,n)` extend the environment *e* with a list of variables *m* and a list of values *n*; a new environment is returned with the variables and values placed in a table in front of the given environment; the given environment is unmodified
- `create()` returns an empty environment; can be a wrapper for `extend(null,null,null)`

Other useful environment routines are:

- `displayEnv(e)` display environment *e*; print out a readable version of each table in the given environment; if a value is an environment, do not recursively display it
- `displayLocalEnv(e)` Like *displayEnv*, but only prints out the most local table.

Data Structures for Environments

One possibility for implementing environments is to store the variables and values in a particular environment table as two parallel linked lists. Normally, such an approach is not ideal due to the linear time access of of variable values and the possibility that the parallel arrays getting out of sync. However, we will see that using parallel arrays allows us to implement function calls very efficiently.

Continuing our standard approach of using *Lexeme* objects for just about everything, we will implement environments with lexemes. We postulate the existence of the lexical functions *cons*, *car*, *cdr.* which are analogues to the eponymous Scheme functions:

```
function cons(type,carValue,cdrValue) { return this; }
function car(cell) { return cell.carValue; }
function cdr(cell) { return cell.cdrValue; }
```

In addition, we will add the functions *setCar* and *setCdr*, which destructively modify the head and tail of a list, respectively, and *type*, which returns the type of a cons cell:

```
function setCar(cell,value) { cell.carValue = value; }
function setCdr(cell,value) { cell.cdrValue = value; }
function type(cell) { return type; }
```

Implementing Environment Functions

By implementing a table with with a cons cell version of a linked list and an environment as a cons cell version of a linked list of tables, the basic environment routines are rather straightforward to implement.

Creating environment structures

After a parse tree is built from the source code, one *evaluates* the tree. When variable and function definitions are evaluated, changes to the current scope occur. In other words, the environment that is holding the current scope of the execution point is populated with variables and their values. Prior to that, however, there are no variables and no values. Therefore, at the start, one needs an empty environment to populate.

Because an environment structure is a list of parallel lists, an empty environment structure has no bindings and thus can be represented by two empty lists joined together into a list of one item. An easy way to do this is to extend the null environment with a null variable list and a null value list:

```
function createEnv()
    {
    return extendEnv(null,null,null);
    }
```

We will implement *extendEnv* in the next section.

Extending an environment

The *extendEnv* function is used to implement function calls. The list of variables passed to the function is the list of formal parameters of the function being called. The list of values passed is the evaluated list of arguments in the function call. The environment being extended is the environment under which the function being called was defined.

To extend a given environment, we simply make a new table out of the given variable and value lists and *cons* it onto the front of the environment: step is performed for a function call;

```
function extendEnv(env,variables,values)
    {
    return cons(ENV,makeTable(variables,values),env);
    }
```

Note that we tag the returned structure with an `ENV` type, so that we can easily identify environment structures. The *makeTable* function is just as simple:

```
function makeTable(variables,values)
    {
    return cons(TABLE,variables,values);
    }
```

Looking up and updating the value of a variable

When code being evaluated references a particular variable, the value of that variable in the current scope needs to be retrieved or updated, depending on the type of reference. When a value is needed, this is referred as an *r-value*. When a value needs to be updated, this is referred to as an *l-value*.

Finding the *r-value* or the *l-value* of a variable simply means walking the variables list in the first table of a given environment until the variable is found. Its value is in the analogous position in the parallel list of values. If the variable is not found in that table, we scan the parallel lists in the second table, and so on. Using that strategy, the *lookupEnv* routine can be implemented as:

```
function lookupEnv(env,variable)
    {
    while (env != null)
        {
        var table = car(env);
        var variables = car(table);
        var vals = cdr(table);
        while (variables != null)
            {
            if (sameVariable(variable,car(variables)))
                return car(values);
            //walk the lists in parallel
            vars = cdr(variables);
            vals = cdr(values);
            }
        env = cdr(env);
        }

    Fatal("variable ",variable," is undefined");

    return null;
    }
```

The *update* function is similar, only *setCar* is used to set the appropriate car pointer of the values list.

Consider the code statement:

```
    x = 2 * y;
```

Since the variable *y* is on the right-hand side of an assignment operator, it is an *r-value* and thus the *lookupEnv* routine is used to retrieve its value. Conversely, the variable *x* is on the left-hand side of the assignment and is therefore an *l-value*. Once the value of 2 * y is calculated, the *updateEnv* routine is used to change the value of *x* from its previous value. Technically speaking, the *l-value* of a variable is the location of its value in the values list.

Defining a Variable

A variable is inserted into the local environment any time a simple variable is declared or a function defined. Note that the local environment is represented as the first two parallel lists in a list of tables. This makes the task quite easy:

```
    function insert(variable,value,env)
        {
        var table = car(env);
        setCar(table,cons(GLUE,variable,car(table)));
        setCdr(table,cons(GLUE,value,cdr(table)));
        return value;
        }
```

Notice that the new variable and its value are put on the front of the parallel lists, since this placement is much faster than placing them at the end of the parallel lists. The `GLUE` type is used when there is no need to give the *cons* cell a specific type. The added advantage of this placement is that redefinitions in the same scope shadow previous definitions.