CS403: Programming Languages

# Notes on Building and Manipulating Environments

## Implementing Environments and Scope

Recall that the purpose of an environment is two fold: to hold the bindings between variables and their current values and to implement scope.

There are five basic operations on environments:

- create
- lookup
- update
- insert
- extend

One possibility for implementing environments is to store the variables and values in a particular environment as two parallel lists. Normally, such an approach is not ideal due to 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 (as before) the existence of *cons*, *car*, *cdr*, *setCar* and *setCdr* functions. The basic environment routines are rather straightforward. Since environments need to point to other environments, we will model an environment structure as a list of parallel lists. The first two lists in an environment structure correspond to the variables and values in the innermost scope. The next two lists will be the variables and values in the nearest outer scope, and so on.

## Creating environment structures

Because an environment structure is a list of parallel lists, an new environment structure has no bindings and thus can be represented by two empty lists in a list:

```
function create()
    {
    return cons(ENV,null,cons(VALUES,null,null));
    }
```

Note that we are using a typed *cons* function. It returns a lexeme whose left pointer is the second argument and whose right pointer is the third argument and whose type is the first argument.

Chapter 4 of the textbook uses a slightly different approach. It treats a scope/table as single node which glues together the list of variable and the lists of values. These scopes or tables are then collected into a linked list. Under this scenario, *create* becomes:

```
function create()
    {
    return cons(ENV,cons(VALUES,null,null),null);
    }
```

Either method works well, but the former is a little bit simpler.

## Looking up and updating the value of a variable

Finding the value of a variable simply means looking up a variable/value in the first two lists of an environment structure. If it is not there, we lookup the variable in the next two lists and so on:

```
function lookup(variable,env)
    {
    while (env != null)
        {
        vars = car(env);
        vals = cadr(env);
        while (vars != null)
            {
            if (sameVariable(variable,car(vars)))
                {
                return car(vals);
                }
            vars = cdr(vars);
            vals = cdr(vals);
            }
        env = cdr(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.

Using the Chapter 4 scenario, *lookup* becomes:

```
    function lookup(variable,env)
        {
        while (env != null)
            {
            table = car(env);
            vars = car(table);
            vals = cdr(table);
            while (vars != null)
                {
                if (sameVariable(variable,car(vars)))
                    {
                    return car(vals);
                    }
                vars = cdr(vars);
                vals = cdr(vals);
                }
            env = cdr(env);
            }

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

        return null;
        }
```

As expected, the differences between the two versions are slight.

## Inserting a new variable into the local environment

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 environments. This makes the task quite easy:

```
    function insert(variable,value,env)
        {
        setCar(env,cons(JOIN,variable,car(env)));
        setCar(cdr(env),cons(JOIN,value,cadr(env)));
        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 JOIN 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.

Under the Chapter 4 scenario, *insert* becomes:

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

## Extending an environment

The last environment routine is *extension*. This is the step is performed for a function call; a new environment is created, populated with the local parameters and values, and finally pointed to the defining environment. The populating step is performed by *cons*-ing on a list of variables and a list of values onto the environment list containing the defining environment:

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

Under the Chapter 4 scenario, *extend* becomes:

```
function extend(variables,values,env)
    {
    return cons(ENV,cons(VALUES,variables,values),env));
    }
```

Note that *create* can be written in terms of *extend*:

```
function create()
    {
    return extend(nil,nil,nil);
    }
```