CS403: Programming Languages

Notes on Undefined Variables

Undefined Variables

Suppose you have an undefined variable in the source code your language intrepreter is evaluating. Typically, what happens is when the evaluator requires the value of a variable, it calls the environmental *lookup* routine. The *lookup* function signals an error when it cannot find the variable in question. We say the undefined variable was detected at *run time*.

An alternative approach is to detect the undefined variable at *parse time*. If your language requires explicit declarations of variables, the task of identifying such variables is rather straightforward. One only needs to detect and store the variable declarations as the parser descends into the source code.

Keeping track of variables declarations

One can use an *environment* to monitor all variable declarations. When a variable is declared, the name of the variable is inserted into the current environment. When a function is defined, the name of the function is placed in the currented environment while an extended environment is used to corral the formal parameters and the local variables encountered when parsing the function body.

To begin, nearly every parsing routine is now passed a *declaration* environment. Consider a parsing function for a simple variable definition:

```
// varDef : VAR ID ASSIGN expr SEMI
function varDef(env)
    {
    var name,value;

    match(VAR);
    name = match(ID);
    match(ASSIGN);
    value = expr(env);
    match(SEMI);

    insert(name,PRESENT,env);       //add ID to the declaration env

    return cons(VARDEF,name,value); //parse tree
    }
```

The environmental *insert* function places the variable name and a dummy value in the local frame of the declaration environment, corresponding to the fact that we just defined a local variable.

Note that we are using a typed *cons* function. It returns a lexeme whose left pointer is the name and whose right pointer is the initialization expression and whose type is `VARDEF`.

The modification for a function definition is similar, except the environment is extended prior to parsing the body:

```
// funcDef : FUNCTION ID OPAREN optParamList CPAREN block
function funcDef(env)
    {
    var name,params,body;
    var xenv;

    match(FUNCTION);
    name = match(ID);
    match(OPAREN);
    params = optParamList();
    match(CPAREN);

    insert(name,PRESENT,env); // add function name to env
```

```
    xenv = extend(params,params,env); //values don't matter, use params

    body = block(xenv);

    return cons(FUNCDEF,name,cons(JOIN,params,body));
    }
```

Note that anytime a block is encountered, the environment needs to be extended since local definitions may occur in the block. The environmental *extend* function takes a list of variables and a parallel list of types, adding a new frame to the existing environment. Note that the original environment is unaffected, a requirement since we cannot have the local variables polluting the calling environment.

As an example, suppose we are parsing the *urp* function:

```
function urp(x,y)
    {
    var z = x - y;
    x * y * z;
    }
```

The name *urp* is inserted into the current declaration environment while the formal parameters and, ultimately, the local variable *z* are inserted into an extension of the declaration environment. Note how this parallels function *calls* when evaluating, wherein the extended environment is populated with the formal parameters and their values. In this case, the value of a formal parameter doesn't matter, we just need it to be present in the extended environment when the body of the function definition is parsed.

Since variable declarations and function definitions (function names and formal parameters) are the only places variables are declared, we are done with the "keeping track *part. What's left is the "checking that variables are previously declared* part.

Checking for previous declarations

The last stage of the implementation of undefined variable detection is to look up every occurrence of a variable (other than in the declarations mentioned previously). If the variable is present in the current environment, everything is copacetic. If not, then no declaration for that variable was encountered on the path to the current spot in the parser and an undefined variable can be signaled.

Since any other mention of a variable is in the *primary* rule, we modify *primary* to make the check:

```
function primary(env)
    {
    var result;

    if (check(INTEGER))
        return match(INTEGER);
    else if (check(STRING))
        return match(STRING);
    else if (check(ID))
        {
        lookup(Pending,env); //throws exception if not found
        result = match(ID);
        if (check(OPAREN))
            {
            //process function call
            ...
            }
        return result;
        }
    //other primaries
        ...

    }
```

At this point, your parser should detect undefined variables.

Mutually recursive functions

A consequence of detecting undefined variables is that all variables be declared before they are used. Sometimes, however, this rule cannot be enforced by the code. Consider two mutually recursive functions:

```
function f(x)
    {
    if (x < 2) { return x; }
    else { return g(x + 1) * x; }
    }

function g(y)
    {
    if (y < 2) { return y; }
    else { return f(y - 2) * x; }
    }
```

When parsing the body of *f*, we encounter a reference to the variable *g*, which has not yet been declared. Likewise, if we place the definition of *g* prior to that of *f* to fix this problem, we generate a second problem. Namely, now *f* is referenced in the body of *g* but now *f* is not yet declared.

Many languages solve this problem by using some sort of *forward* statement. A forward statement is processed just like any other variable declaration:

```
forward g; // g will be declared later

function f(x)
    {
    if (x < 2) { return x; }
    else { return g(x + 1) * x; }
    }

function g(y)
    {
    if (y < 2) { return y; }
    else { return f(y - 2) * x; }
    }
```

Typically, you would insert a forward variable with a distinct value into the environment:

```
// FORWARD ID SEMI
function forward(env)
    {
    var result,name;

    result = match(FORWARD);
    name = match(ID);
    insert(name,newLexeme(FORWARD),env);
    match(SEMI);

    return result;
    }
```

Now, when inserting the actual definition of a forward variable in the local frame, you would update the value of the variable from `FORWARD` to `PRESENT`. By doing the update, you can check for unresolved forward declarations in the local frame after parsing function bodies and after returning from the start rule.