CS403: Programming Languages

Notes on Evaluation of Parse Trees

Evaluation

Once you build parse trees, it's an easy matter to evaluate them. The key is to always evaluate a parser tree in the context of an environment. The environment is necessary to determine the value of variable references as well as to update the values of variables.

The top-level evaluation function

The form of the main evaluating function follows that of a pretty printer; a case exists for every kind of parse tree:

```
function eval(tree,env)
    {
    switch (tree.type)
        {
        //self evaluating
        case NUMBER: return tree;
        case STRING: return tree;
        //find value of variables in the environment
        case VARIABLE: return lookup(tree,env);
        //parenthesized expression
        case OPAREN: return eval(tree.right,env);
        //operators (both sides evaluated)
        case PLUS:
        case MINUS:
        case TIMES:
        case DIVIDES:
        ...
        case GREATER_THAN: return evalSimpleOp(tree,env);
        //AND and OR short-circuit
        case AND:
        case OR: return evalShortCircuitOp(tree,env);
        //dot operator evals lhs, rhs a variable
        case DOT: return evalDot(tree,env);
        //assign operator evals rhs for sure
        //    lhs is a variable or a dot operation
        case ASSIGN: return evalAssign(tree,env);
        //variable and function definitions
        case VAR_DEF: return evalVarDef(tree,env);
        case FUNC_DEF: return evalFuncDef(tree,env);
        //imperative constructs
        case IF: return evalIf(tree,env);
        case WHILE: return evalWhile(tree,env);
        //function calls
        case FUNC_CALL: return evalFuncCall(tree,env);
        //program and function body are parsed as blocks
        case BLOCK: return evalBlock(tree,env);
        default: fatal("bad expression!");
        }
    }
```

Note that the main evaluation function is primarily a dispatch function. It could be re-written in an data-directed fashion:

```
function eval(tree,env)
    {
    var f = getEvalFunction(tree.type);
    if (null?(f))
        throw(EXCEPTION,"no evaluation function for type " + tree.type);
```

```
    else
        return f(tree,env);
    }
```

Specific evaluation functions

The most conceptually difficult sub-evaluation routines are *evalFuncDef* and *evalFuncCall*. However, the code itself is almost trivial. Consider the process of evaluating function definitions. All that needs to be done is create a function object (or *closure*) and insert it into the environment under the function name, as in:

```
function evalFuncDef(t, env)
    {
    var closure =
        cons(CLOSURE,env,
            cons(JOIN,getFuncDefParams(t),
                cons(JOIN,getFuncDefBody(t),
                    null)));
    EnvInsert(env,getFuncDefName(t),closure);
    }
```

One can simplify this function by simply consing on the definition environment to the function definition parse tree, rather than decomposing the tree:

```
function evalFuncDef(t, env)
    {
    var closure = cons(CLOSURE,env,t);
    EnvInsert(env,getFuncDefName(t),closure);
    }
```

The function call evaluator is only slightly more complicated:

```
function evalFuncCall(t,env)
    {
    var closure = eval(getFuncCallName(t),env);
    var args = getFuncCallArgs(t);
    var params = getClosureParams(closure);
    var body = getClosureBody(closure);
    var senv = getClosureEnvironment(closure);
    var eargs = evalArgs(args,env);
    var xenv = EnvExtend(senv,params,eargs);

    return eval(body,xenv);
    }
```

The closure is found by looking up the name of the function being called in the environment. The function call arguments are retrieved as well from the parse tree. Next the formal parameters, the function body, and the static environment are retrieved from the closure. Next the function call arguments are evaluated and, with the formal parameters, are used to extend the static environment. Finally, the body is evaluated under the newly extended environment.

Assuming the body is a block and that a block parse tree is simply a list of *if*s, *while*s, and/or expressions and that the "value" of a block is the value of the last statement executed, then *evalBlock* can be written as:

```
function evalBlock(t,env)
    {
    var result;
    while (t != null)
        {
        result = eval(t . left, env);
        t = t . right;
        }
    return result;
    }
```

Evaluating binary expressions is straightforward, if a bit tedious. First, *evalSimpleOp* dispatches to the proper operator evaluator:

```
function evalSimpleOp(t,env)
    {
    if (t.type == PLUS) return evalPlus(t,env);
    if (t.type == MINUS) return evalMinus(t,env);
    ...
    }
```

The evaluator for plus has to examine the types of the left and right hand sides to determine the kind of lexeme it should produce:

```
function evalPlus(t,env)
    {
    //eval the left and the right hand sides
    var left = eval(t.left,env);
    var right = eval(t.right,env);
    if (left.type == INTEGER && right.type == INTEGER)
        return newIntegerLexeme(left.ival + right.ival);
    else if (left.type == INTEGER && right.type == REAL)
        return newRealLexeme(left.ival + right.rval);
    ...
    }
```

Simple assignment is like any other operator, except the lhs is not evaluated:

```
function evalAssign(t,env)
    {
    //eval the right hand side
    var value = eval(t.right,env);
    EnvUpdate(env,t.left,value);
    return value;
    }
```

However, if objects are provided for in the language, the left-hand side needs to be examined to see if it is a dot operator, as in:

```
a . x = 5;
```

Here is an assignment evaluator that does just that:

```
function evalAssign(t,env)
    {
    //eval the right hand side
    var value = eval(t.right,env);
    if (t.left.type == DOT)
        {
        var obj = eval(t.left.left);
        EnvUpdate(obj,t.left.right,right)
        }
    else
        {
        EnvUpdate(t.left,value,env);
        }
    return value;
    }
```

Note, that the equivalence between objects and environments makes this modification particularly easy.