Programming Languages

Building Parse Trees

Parse trees

We have previously seen that most programming language grammars can be transformed into a *recognizer* via recursive descent. While a recognizer performs a useful task, detecting syntactic errors in the source code, its utility can be greatly enhanced by assembling matched lexemes into structures that abstract the original source. Such abstract structures are called *parse trees*.

Consider the implementation of the following right recursive grammar as a recognizer:

```
expression : unary op expression | unary

op : PLUS | MINUS | TIMES | DIVIDE

unary : VARIABLE | NUMBER | OPAREN expression CPAREN | MINUS unary
```

The function corresponding to *expression* becomes:

```
function expression()
    {
    unary();
    if (opPending())
        {
        op();
        expression();
        }
    }
```

Note that while the *expression* function enforces the proper structure of an expression, it does not save the lexed source code; all the lexemes are thrown away. In order to save the lexemes and to combine into a structure representative of the input, we will now take advantage of the fact that *match* returns lexemes. We will convert all of the recursive descent functions to return the structure that function has recognized, in the form of a tree of lexemes. We also make sure the tree being built is tagged with the kind of structure that was recognized. We do this by using a special lexeme as the root of the tree.

It is assumed the lexeme objects are outfitted with left and right pointers so that they can be assembled into binary trees. Here is the updated *expression* function.

```
function expression()
    {
    var u;

    u = unary();
    if (opPending()) //a binary expression
        {
        var tree = new Lexeme(BINARY); //this is the tag
        tree.left = op();
        tree.right = new Lexeme(GLUE); //an interior node
        tree.right.left = u;
        tree.right.right = expression();
        return tree;
        }
    else //a unary expression
        return u;
    }
```

Note that the calls to non-terminals and pending functions are preserved in exactly the same order as before. Now, however, the *expression* function returns a tree whose root, in the case of a complex expression, is a lexeme of type `BINARY` and whose left and right subtrees hold the operator and the operands of the expression, respectively. In the case of a simple expression (*unary* only), the tree returned by the *unary* function is returned. The *op* and *unary* routines become:

```
function op()
    {
    var tree;
    //the tree is the operator that was found
    if (check(PLUS)) tree = advance();
    else if (check(MINUS)) tree = advance();
    ...
    else if (check(ASSIGN)) tree = advance();
    ...
    else if (check(LESS_THAN)) tree = advance();
    else tree = match(LESS_THAN_OR_EQUAL);

    return tree;
    }

function unary()
    {
    var tree;

    if (check(VARIABLE)) tree = advance();
    else if (check(INTEGER)) tree = advance();
    else if (check(OPAREN))
        {
        tree = match(OPAREN); //tree is tagged OPAREN
        tree.left = null;
        tree.right = expression();
        }
    else //unary minus
        {
        match(MINUS);
        tree = new Lexeme(UMINUS); //tag the tree
        tree.left = null;
        tree.right = unary();
        }

    return tree;
    }
```

Note that the lexeme type in the unary minus case was renamed to reflect that the binary MINUS sign was overloaded at the lexical level. In general, a parsing function has the flow-of-control as the associated recognizing function, but returns a parse tree assembled from the *good stuff* in the source code. Usually, the *good* stuff is the minimal set of lexemes needed to reconstruct the source. For example, consider a function definition:

```
functionDef : FUNCTION VARIABLE OPEN_PAREN parameterList CLOSE_PAREN block
```

As a parsing function, a reasonable implementation of *functionDef* would be:

```
function functionDef()
    {
    var tree;
    tree = match(FUNCTION);             //tag the tree
    tree.left = match(VARIABLE);
    match(OPEN_PAREN);                  //no need to save
    tree.right = new Lexeme(GLUE);
    tree.right.left = optParameterList();
    match(CLOSE_PAREN);                 //no need to save
    tree.right.right = block();
    return tree;
    }
```

Note that the parentheses enclosing the parameter list were not saved in the tree that was returned. This is because the original definition in the source code can be recreated with the parts that were saved. Punctuation is usually not preserved in parse trees.

Once the recognizer is completely converted to a parser, the top-level rule (the first rule called) returns a parse tree that represents the entire source code program. This is not the only way to handle parse trees, but it is the simplest way.

One converts a recognizer to a parser by ensuring every recursive descent function returns a parse tree. The shape of the parse tree is irrelevant, as long as the important lexemes are stored in the tree. It is important for a function that builds a parse tree to tag the tree it returns with some indication of what the tree represents.

Sometimes, however, a recursive descent function merely returns a tree that was assembled by a different function. The function *optParameterList* is an example of this:

```
function optParameterList()
    {
    if (parameterListPending())
        return parameterList();
    else
        return null; //NULL signifies an empty list
    }
```

If one uses `null` as a sentinel value for *empty*, then there is no need for *optParameterList* to build its own tagged parse tree.

Building a pretty printer

Speaking of recreating the source code from parse trees, a program known as a *pretty printer* does just that. Recall that lexemes are objects that can hold any kind of value in the language and that the component *ival* stores the value of integer lexemes and that the component *sval* stores the value of variable and string lexemes. One possible way of implementing the pretty printer is through a switch statement that steps through all the possible lexeme types:

```
function prettyPrint(tree)
    {
    switch (tree.type)
        {
        case INTEGER  { print(tree.ival);}
        case VARIABLE { print(tree.sval); }
        case STRING   { print('\"', tree.sval, '\"'); }
        case OPAREN
            {
            print("(");
            prettyPrint(tree.right);
            print(")");
            }
        case UMINUS
            {
            print("-");
            print(tree.right);
            }
        case BINARY
            {
            prettyPrint(tree.right.left);  //the left operand
            print(tree.left);              //the operator
            prettyPrint(tree.right.right); //the right operand
            }
        ...
        else { fatal("bad expression!"); }
        }
    }
```

This pretty printer is pretty basic and not very pretty. In fact, it prints an entire expression on a single line, regardless of its size. The pretty printing code for a function definition might look like:

```
case FUNCTION
    {
    print("function ");
    prettyPrint(tree.left);
    print("(");
    prettyPrint(tree.right.left);
    print(") { ");
    prettyPrint(tree.right.right);
    println("}");
    }
```

More sophisticated pretty printers have additional arguments, the most useful of which is an indentation level. However, getting a pretty printer to print out beautiful looking code is a rather tedious process, involving much trial and error.

Even an ugly pretty printer is useful, as it can tell you whether your parse tree correctly represents the original source code. Moreover, it can be used to implement print statements in your language; if your pretty printer can handle every kind of parse tree in your language, a print function based upon your pretty printer will be able to do the same.