CS403: Programming Languages

Notes on Parse Trees

Parse trees

Recall that the implementation of a grammar is called a *parser*. Recall also that a parser which skims through the input in order to see that the input is made of good sentences is designated a *recognizer*. A recognizer can be extended to format the input in a nicer or standard fashion. Such a program is called a *pretty printer*. However, in order to pretty print,the structure of the input sentences must somehow be preserved. An easy way to do this is to build a parse tree.

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

```
expression : primary op expression | primary

op : PLUS | MINUS | TIMES | DIVIDE

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

The function corresponding to *expression* is..

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

Note that while the *expression* function enforces the proper structure of an expression, it does not save the 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 parsing functions to return the structure that function has recognized, in the form of a tree of lexemes. It is assumed the lexem 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 tree;

    tree = primary();
    if (opPending())
        {
        var temp;
        temp = op();
        temp.left = tree;
        temp.right = expression();
        tree = temp;
        }

    return tree;
    }
```

The *expression* function now returns a tree whose root, in the case of a complex expression, is the operator and whose left and right subtrees hold the left-hand side and the right-hand side of the binary operation, respectively. In the case of a simple expression (primary only), the tree returned by the primary function is returned directly. The *op* and *primary* routines simply become:

```
function op()
    {
```

```
    return match(ANYTHING); //type ANYTHING matches all operator types
    }

function primary()
    {
    var tree;

    if (check(VARIABLE))
        {
        tree = match(VARIABLE);
        }
    else if (check(NUMBER))
        {
        tree = match(NUMBER);
        }
    else if (check(OPAREN))
        {
        tree = match(OPAREN);
        tree.left = null;
        tree.right = expression();
        }
    else //unary minus
        {
        tree = match(MINUS);
        tree.type = UMINUS; //rename!
        tree.left = null;
        tree.right = primary();
        }

    return tree;
    }
```

Note that the lexeme type in the unary minus case was renamed to reflect that the MINUS sign was overloaded at the lexical level.

Building a pretty printer

A pretty printer simply reformats the input. Since we've saved the input and its structure in a parse tree, it is a relatively simple task to regenerate the original input. Recall that lexemes are objects that can hold any kind of value in your 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 NUMBER   { print(tree.nval);}
        case VARIABLE { print(tree.sval); }
        case STRING   { print('\"', tree.sval, '\"'); }
        case OPAREN
            {
            print("(");
            prettyPrint(tree.right);
            print(")");
            }
        case UMINUS
            {
            print("-");
            print(tree.right);
            }
        case PLUS
            {
            prettyPrint(tree.left);
            print(" + ");
```

```
            prettyPrint(tree.right);
            }
        .
        .
        .
        else { print("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. 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.