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 : unary op expression | unary

op : PLUS | MINUS | TIMES | DIVIDE

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

The function corresponding to `expression` is:

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

```javascript
function expression()
{
    var u,o,e;
    u = unary();
    if (opPending())
    {
        o = op();
        e = expression();
        return cons(o.type,u,e);
    }
    else
    return u;
}
```

The `cons` function constructs a new lexeme with the first argument giving the type of the lexeme, the second argument giving the left pointer and the third argument giving the right pointer. Thus, 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 (`unary` only), the tree returned by the `unary` function is returned directly. The `op` and `unary` routines simply become:

```javascript
function op()
{
    return match(ANYTHING); //type ANYTHING matches all operator types
}
```
function unary() {
    var tree;

    if (check(STRING))
        return match(STRING);
    else if (check(NUMBER))
        return match(NUMBER);
    else if (check(OPAREN))
        {
            match(OPAREN);
            tree = expression();
            return cons(OPAREN, null, tree);
        }
    else if (varExpressionPending())
        return varExpression();
    else // unary minus
        {
            match(MINUS);
            tree = unary();
            return cons(UMINUS, null, u);
        }
}

Note that the lexeme type in the unary minus case was renamed to reflect that the MINUS sign was overloaded at the lexical level. The cons routine is easily implemented as:

    function cons(type, left, right) {
        var p = new Lexeme(type);
        p.left = left;
        p.right = right;
        return p;
    }

The name cons is shorthand for construct.

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 INTEGER { print(tree.ival);}
            case REAL { print(tree.rval);}
            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.