Recursive Descent Parsing

This is an **OPTIONAL** handout, for students who would like to know a little more about the technique of recusrsive descent parsing. An informal introduction to some of the terminology and ideas is provided, using homework 3 as an example.

You are **NOT** required to read this in order to be able to do the homework. In case you are having problem understanding the homework, please contact the consultants, TAs or post your query to the newsgroup. This is simply a little extra information for the curious.

What's parsing ?

There are many occasions where one has to read some text, understand it's structure, and perform some action accordingly. For example, the C++ compiler has to read a program (a piece of text), understand it (that is, identify the keywords, variables, methods and class structures) and output machine code (in a .exe file, typically) that a computer can execute. Another example is the current homework, where your program has to read an infix expression, understand it (that is, identify the variables, operators, functions calls, and how they relate to one another) so that it may evaluate the expression if it is given the values for the variables. Both the C++ program, and the infix expression have a structure of their own. The process of understanding a piece of text based on it's structure, (n order to perform some action) is called parsing.

What are tokens ?

When we're parsing text, it is convenient to deal with *words*, rather than process text character by character. The first stage of parsing usually involves breaking up the text to be parsed into words, or blocks of text that have some meaning. We call these blocks of text *tokens*. A parser for a Java program would first break the program into a sequence of tokens, which would be Java's reserved words (like public, int, class, throws, if) method names, variable names, etc. In the current homework, the tokens include function names (like sin, cos, max, min), numbers, parentheses and operators (+,-,*,/).

An object that breaks text into the desired tokens is called a tokenizer. For your homework, we have provided a class ExpressionTokenizer. This breaks the string representing the infix expression into a sequence of tokens. For example, the string "(1+x) * 2 -cos(y/ 3)" will be broken into { "(", "1", "+", "x", ")", "*", "2", "-", "cos", "(", "y", "/","3", ")" }. The irregular spacing between parts of the epression does not have any effect.

Parsing: Combining Tokens Meaningfully

We can group together a sequence of tokens into a meaningful item. For example,

```
if (a == 0)
        return 5;
```

is a group of Java tokens ("if", "(", "a", ..., "return", "5", ";") which together define a meaningful Java statement. Grouping of tokens is governed by well defined rules. In the above example, we have to group the tokens according to Java's syntax.

In your homework, your program should group tokens according to standard arithmetic rules. Tokens could be grouped into expressions, terms and factors. The diagram below shows how the tokens fit into their categories (expressions, terms, factors, numbers, function calls, etc.)