

## ASSIGNMENT #2

Due Tuesday, January 20, 2004

Consider the following extended BNF grammar for a subset of the C++ programming language, called MicroC++.

```
program          ::=  #include <iostream.h>
                   #include "list.h"
                   { declaration ; }
                   function-definition { function-definition }
function-definition ::=  type function-identifier ( [declaration { , declaration } ] )
                   { { declaration ; } { statement } return expression ; }
type              ::=  int | list
declaration       ::=  type object-identifier
statement         ::=  { statement { statement } }
                   | object-identifier = expression ;
                   | if ( expression ) statement [else statement ]
                   | while ( expression ) statement
                   | cout << expression ;
expression        ::=  term | expression binary-op expression
term              ::=  primary-expression | unary-op term
primary-expression ::=  object-identifier | integer | ( expression )
                   | function-identifier ( [ expression-list ] )
                   | primary-expression . cons ( expression )
                   | primary-expression . head ( )
                   | primary-expression . tail ( )
                   | nil
expression-list   ::=  expression { , expression }
```

**Syntactic and Semantic Conventions** The keywords and the token symbols in MicroC++ are in bold. Note that MicroC++ has symbols { and }, which are distinguished from grammar metasympols { and }, respectively, by underlining. The unary operators are +, - and ! (negation) Binary operators obey the customary precedence rules, from highest to lowest:

```
multiplicative   *, /
binary additive  +, -
relational        ==, !=, <, >, <=, >=
boolean          &&&, ||
```

Assume that an identifier can only contain letters (only alphabetic characters), digits, and underscores (-) with the restrictions that it must begin with a letter, cannot end with an underscore and cannot have two consecutive underscores. For example, give\_2\_Joe, tell\_me and A45Asm3 are valid identifiers, but 6gh, two\_\_bad, and no\_end\_ are not. object-identifiers and function-identifiers are the same lexical/syntactic items as identifiers but have the semantics given by the appropriate qualifier. integer is an unsigned integer. Assume that comments are indicated by being preceded by //.

1. Determine the set of tokens which a lexical analyzer would need to recognize.
2. Design and implement a lexical analyzer procedure to read a source program in the above language and print the next token in the input stream. If the token detected is a valueless token, such as a keyword, then it is sufficient to print only the keyword. If it has a value, then both the token type and lexeme should be printed.
3. You will be given several MicroC++ programs with which to test your lexical analyzer. These will be located on the class WWW page and will be of the form Test1.cpp, Test2.cpp, etc.

**Suggestions:**

1. Construct a token class to represent the token data structure, including a method to print a token.
2. Use the JLex tool to automatically construct a lexical analyzer in Java from a set of regular expressions specifying tokens. This can interface with your token class to return a token to the main program which then prints the token.