CS 132: Compiler Design, Fall 2007

# Assignment 2
# Lexing and Parsing Practice

Due: Wednesday, September 19, 11:59pm

## Goals for this Assignment:

1. Writing non-trivial lexers and parsers
2. Using LR and LL-based parser generators.

E-mailed questions about this assignment should be sent to `cs132help@cs.hmc.edu`.

1. Use `svn copy` to make a full copy of the `src/a2` directory. It contains code for *two* lexer/parser pairs for the original Assignment 1 language. The former (files named `-LR`) uses the ml-lex and ml-yacc tools, while the latter (files named `-LL`) uses the ml-ulex and ml-antlr tools. You can compile the code in SML/NJ using `CM.make "arith-LR.cm";` or `CM.make "arith-LL.cm";` as appropriate.

   Once either of these has been compiled and loaded, you can use the function `Toplevel.parse` to get back abstract syntax, and `Toplevel.compile` to parse a file and write out a `.j` file.

2. Complete the file `translate.sml` by writing functions to recursively traverse the abstract syntax of a program (as defined in `absyn.sml`) and to generate the appropriate JVM code, again as a list of strings. This should be easy given your experience doing Assignment 1.

   As you might expect, the JVM supports both an `imul` operation to multiply two integers on the stack and an `ineg` operation to negate an integer on top of the stack.

3. Extend the lexers and parsers to handle the following extensions:

   - Your parser and translator must handle both `while` loops and parenthesized arithmetic expressions using infix operators instead of `add` and `sub` keywords. (Those two keywords should be removed.)
   - Expressions should permit multiplication.
   - Unary negation should bind most tightly, followed by multiplication, followed by addition and subtraction. All three binary operators should be left associative.
   - The lexer should skip (treat as whitespace) line comments, start with `//` and run to the end of the line[1].

   In contrast to your code for Assignment 1, your parser should return an abstract syntax tree (i.e., the result of the whole parse should be a value of type `Absyn.program`). See `absyn.sml` for the definition of the abstract syntax.

[1] Warning: contrary to its documentation, the `ml-ulex` tool lets `.` match any character *including* newline. So the regular expression `"//".*` for a line comment in the `-LL` lexer would treat the whole rest of the program as a line comment.