CS 3323 - Principles of Programming Languages
Assignment 2
Fall 2020

The objective of this programming assignment is to build a fully functional parser. Instructions are provided below.

First, download the files grammar.y, scanner.yy, inputs.tar.gz (decompresses to directory **inputs**), Makefile and driver.c from the assignment directory space. Then, perform the necessary modifications to the grammar's file to accept/reject the example programs provided. You should only work on the grammar file.

To decompress the tarball with the test files run in a command line:

```
tar xvzf inputs.tar.gz
```

You will then find a directory named "inputs" in your working directory.

The assignment is due on Friday October 16th, 2020, 11:59pm. Files must be uploaded by then. If you worked with a group of classmates for the first assignment, you should continue to work with them in this one. Late policy deduction applies.

You must complete the production rules of grammar.y. While a description of each rule is provided below, you should also use the input files provided to further understand the syntactic structure that is being requested. In some cases, partial implementations of the requested productions are also given, but you will not be able to test them until the grammar is completed.

1. (0.5pt) Complete the production corresponding to the **varlist** non-terminal (Line 145 in grammar.y), which is used in the production of the non-terminal **read** (Line 139 in grammar.y). **varlist** should produce a comma-separated list of variable references (**varref**). The list of variable references should be of at least length one.

2. (0.5pt) Complete the production corresponding to the **expr_list** non-terminal (Line 149 in grammar.y). It should produce a comma-separated list of arithmetic expressions (**a_expr**). The list of arithmetic expressions should be of at least length one.

3. (1.5pt) Define three productions for the non-terminal **l_fact** (Line 124 in grammar.y):

   - a left-recursive rule producing comparisons of arithmetic expressions (**a_expr** non-terminal). It should use the **oprel** non-terminal already defined.
   - a single arithmetic expression.
   - A logical expression in parenthesis (**l_expr** non-terminal).

4. (1pt) Define two productions for the **varref** non-terminal (Line 112 in grammar.y) that match the below description:

   - A variable reference can be the T_ID token.
   - A variable reference can be a left-recursive list of arithmetic expressions delimited by '[' and ']'. The recursion terminates with the T_ID token (See above description).

5. (2pt) Define five productions for the non-terminal **a_fact** (Line 105 in grammar.y) based on the following description:

   - An **a_fact** can be a variable reference (non-terminal **varref**).

- The token T_NUM.
- A literal string (token T_LITERAL_STR).
- The non-terminal **a_fact** preceded by the T_SUB token (Note: Do not use '-').
- A parenthesized arithmetic expression.

6. (2pt) Complete the control-flow constructs (Lines 74–91 in grammar.y). Observe that a statement list surrounded by the tokens T_BEGIN and T_END is also a statement. The non-terminal **l_expr** must be used for representing logical expressions. Use test cases for*.smp, if*.smp, repeat*.smp and for*.smp.

   - **foreach**: Complete the partially-defined production. See input cases for[1-4]_pass.smp.
   - **repeat-until**: Define it as a list of statements. Use the non-terminal **stmt_list**). The list must be delimited by the tokens T_REPEAT and T_UNTIL. The controlling condition should use the **l_expr** non-terminal. Do not add parentheses.
   - **while**: The T_WHILE token followed by a logical expression and any statement.
   - **if-then/if-then-else**: The T_IF token followed by a logical expression (non-terminal **l_expr**). The true branch should be a statement preceded by the T_THEN token, whereas the T_ELSE branch can either be empty or start with the T_ELSE token followed by a statement.

For convenience, a Makefile is provided, but you are not required to use it.
To rebuild the binary (**simple.exe**) run: make all
To test a single input file, run: ./simple.exe < inputfile.smp
Several online resources can be found in the web, for instance:

- `https://www.gnu.org/software/bison/manual`
- `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#multiplicative-expression`

More resources can be found by searching for the key terms: yacc/bison parser generator.
Do not change the driver file, nor the scanner.yy files. Do not print anything to the output.
Every student should upload a single file named: ABCDEFGHI.y, where ABCDEFGHI is the 9-digit code identifying the student (not the 4+4).