CMPSC 470 **Project 2** Due: Thursday, October 31, 2018

Syntax Checker (100 points)

Syntax checker is a program that checks syntax error in a program. The syntax checker reads a source program, checks if there is any syntax error, and outputs error messages describing any syntax errors found.

In this assignment, you should build a simple syntax checker implemented using java. Following describes tasks of your program:

- Your program should read an input *mini-C* source program using **`jflex`** for tokenizing.
- Your program should check if the input program follows the (given subset) of *mini-C* grammar. Your program should parse the input program using the recursive descent parser or the recursive predictive parser (recommended), discussed in class. The subset of *mini-C* grammar will be given at the end of this document, in the form of CFG.
- If there is **no** syntax error, then your program should print "**`Success: no syntax error found.`**";
  If there is **any** syntax error(s), then your program should print (i) the first error message with line number and (ii) "**`Error: there exists syntax error(s).`**".

For example, let you have the `syntax-checker` java program, and the following sample input *mini-C* programs:

succ_01.minc

```
int main(int a)
{
    int x;
    x = x + 1;
}
```

fail_01.minc

```
int main(int a)
{
    int x;
    x == x + 1;  // There is a syntax error
}
```

Running `syntax-checker` program with the above *mini-C* programs should print the following outputs on console:

```
> java syntax-checker succ_01.minc
Success: no syntax error found.
>
```

```
> java syntax-checker fail_01.minc
Syntax error: "=" is expected instead of "==" at line 4.
Error: there exists syntax error(s).
>
```

Note that the `syntax-checker` does not check semantic errors, such as variable type, return type, or declaration of variable.

There are two types of syntax errors that your `syntax-checker` can identify in this assignments. Your program must distinguish the two different error messages.

Case 1
 	
Accurate syntax error can be easily identified since "if" and "(" should come together.
	`int main(int a)`
	`{`
	`  if abc ) // "(" must be followed after if`
	`  { } else { }`
	`}`
	`Syntax error: "(" is expected instead of "abc" at line 3.`
	`Error: there exists syntax error(s).`

Case 2
 	
It is more difficult to identify accurate error message.
	`int main(int a)`
	`{`
	`  int x;`
	`  x = x + 1;`
	`  int y;`
	`}`
	`Syntax error: There is a syntax error at line 5.`
	`Error: there exists syntax error(s).`

Test cases and points

Totally 38 test cases (*mini-C* programs) will be provided at https://turing.cs.hbg.psu.edu/cmpsc470/proj2-testcases.zip in our course website. 16 *mini-C* programs whose filenames start with "`succ_`" will not have syntax error; 22 *mini-C* programs whose filenames start with "`fail_`" will have syntax error(s). Additionally, outputs of all 38 test cases will be provided, whose name start with "`output_`".

You will get following points per each test program when your `syntax-checker` is implemented **as recursive predictive parser**:

- 1 point if your `syntax-checker` correctly prints "`Success: no syntax error found.`";
- 1 point if your `syntax-checker` correctly prints "`Error: there exist syntax errors.`"; and
- 1 point if your `syntax-checker` correctly prints the first error message, such as "`Syntax error: "=" is expected instead of "==" at line 4.`"

From the initially given 38 test cases, you will get up to $16 + 22 * 2 = 60$ points. The grader will add additional test cases, in order to check if your `syntax-checker` program works correct. Finally, you will get 100 points from all test cases. Make sure your messages be exactly same to the sample outputs; grader may use a file comparison program such as `diff`.

If your `syntax-checker` is implemented **as recursive descent parser**, then you will get 0.7 points in each bullet and you will get 70 points from all test cases.

Note that you **may lose some points:**

- if your program does not terminate or causes runtime errors, such as segmentation fault; or
- if your program has odd variable or function names (such as bad words).

What to submit:

- Submit one zip file containing following files to **canvas by 11:59:59 PM, Wednesday, October 31, 2018**.
- **Readme file** describing how to compile your syntax checker program and how to run it. Grader will recompile your jflex file(s).
- Your own **jflex** and **java** source files that implements the syntax checker program.

The *mini-C* grammar:

The following describes the minimal subset of ***mini-C*** grammar:

```
1.  program         -> decl_list
2.  decl_list       -> decl_list  fun_decl | epsilon
3.  type_spec       -> "int"
4.  fun_decl        -> type_spec IDENT "(" params ")" compound_stmt
5.  params          -> param_list | epsilon
6.  param_list      -> param_list "," param | param
7.  param           -> type_spec IDENT
8.  stmt_list       -> stmt_list stmt | epsilon
9.  stmt            -> expr_stmt | compound_stmt | if_stmt | while_stmt
10. expr_stmt       -> IDENT "=" expr ";" | ";"
11. while_stmt      -> "while" "(" expr ")" stmt
12. compound_stmt   -> "{" local_decls stmt_list "}"
13. local_decls     -> local_decls local_decl | epsilon
14. local_decl      -> type_spec IDENT ";"
15. if_stmt         -> "if" "(" expr ")" stmt "else" stmt
16. arg_list        -> arg_list "," expr | expr
17. args            -> arg_list | epsilon
18. expr            -> expr "+" term | term
19. term            -> term "==" factor | term "*" factor | factor
20. factor          -> "(" expr ")" | IDENT | IDENT "(" args ")" | INT_LIT
```

In the above, italicized words represent non-terminals (such as *program* and *type_spec*), *program* is the starting symbol, words or symbols enclosed by quotation marks are keywords and symbols (such as "int" and "{"), respectively, and

- *epsilon* represents the empty string,
- **INT_LIT** represents positive integers,
- **IDENT** represents C language identifiers, which start with lowercase letter or capital letter or '_'.

Comments (a string that starts with "//" and ends at the end of line), new line (\n), and whitespaces([\t\r]+) must be ignored in lexical analyzer implemented using **jflex**.

In the *mini-C* grammar,

- The productions 1-2 define the start state that list functions;
- The productions 3-7 define the function;
- The productions 8-15 define the list of statements, if-statement, while-statement, and expression statement, and compound-statement (that can declare variables and other statements in between curly brackets);
- The productions 16-17 defines the list of arguments that are passed to a function call;
- The productions 18-20 define algebraic expression;

The above grammar contains very minimal set of *mini-C* grammar. In next project, the semantic checker will be developed using **BYacc/J** with including additional grammars of *mini-C*.

After eliminating left-recursion and performing left-factoring of the subset of *mini-C* grammar, the updated grammar becomes as follows:

```
1.   program         -> decl_list
2.   decl_list       -> decl_list'
3.   type_spec       -> "int"
4.   fun_decl        -> type_spec IDENT "(" params ")" compound_stmt
5.   params          -> param_list               | epsilon
6.   param_list      -> param param_list'
7.   param           -> type_spec IDENT
8.   stmt_list       -> stmt_list'
9.   stmt            -> expr_stmt                | compound_stmt | if_stmt | while_stmt
10.  expr_stmt       -> IDENT "=" expr ";"       | ";"
11.  while_stmt      -> "while" "(" expr ")" stmt
12.  compound_stmt   -> "{" local_decls stmt_list "}"
13.  local_decls     -> local_decls'
14.  local_decl      -> type_spec IDENT ";"
15.  if_stmt         -> "if" "(" expr ")" stmt "else" stmt
16.  arg_list        -> expr arg_list'
17.  args            -> expr arg_list'           | epsilon
18.  expr            -> term expr'
19.  term            -> factor term'
20.  factor          -> IDENT factor'            | "(" expr ")" | INT_LIT
21.  decl_list'      -> fun_decl decl_list'      | epsilon
22.  param_list'     -> "," param param_list'    | epsilon
23.  stmt_list'      -> stmt stmt_list'          | epsilon
24.  local_decls'    -> local_decl local_decls'  | epsilon
25.  arg_list'       -> "," expr arg_list'       | epsilon
26.  expr'           -> "+" term expr'           | epsilon
27.  term'           -> "==" factor term'        | "*" factor term' | epsilon
28.  factor'         -> "(" args ")"             | epsilon
```