CMPSC 470

Project 1

Due: Wednesday, February 5, 2024

Tokenizer (100 points)

In this course, a compiler for mini-C (a mixture of mini-C and Julia language) will be built through a sequence of projects. Its first step is building a tokenizer for the mini-C language. Later, this tokenizer will be replaced with a lexical analyzer (or lexer) that is generated using `jflex` in the next assignment.

Tokenizer is the first phase of the compiling process. Tokenizer 1) takes a program source from an input stream, 2) breaks the source into a sequence of lexemes that are meaningful units of words for tokens given regular definitions, 3) generates tokens, each of which is a pair of a token name and its attribute, and 4) passes them to its caller (parser here) repeatedly.

Startup files:

You can download the startup files and 8 test inputs/outputs from the course website:

- https://h3turing.cs.hbg.psu.edu/cmpsc470/proj1-minic-tokenizer-startup.zip

The startup files are composed of the following `flex` and `java` files:

1. `Lexer.java`: a sample tokenizer file that splits a given input to a sequence of tokens;
2. `Parser.java`: a sample parser file that prints token information on the console;
3. `Compiler.java`: a sample compiler file that calls the parser;
4. `ParserVal.java`: the class that contains each token's attribute information. *Do not modify this class*;
5. `Program.java`: the simple class that has the main function. *Do not modify this class*.
6. `test`*k*`.minc, testsolu`*k*`.txt`: the *k*-th test min-C file and its corresponding solution file.

Project description:

In this project, you have to implement a mini-C tokenizer using (a) double buffering whose buffer size must be 10 and (b) a transition diagram of the regular definitions. As a result, your program should generate and print a list of tokens on the console when `Lexer` returns those tokens to `Parser`.

To finish this assignment,

1. you should update `Lexer.java` that returns all tokens for mini-C compiler (excluding white spaces) to `Parser`,
2. you should update `Parser.java` that prints tokens on the console in a form explained in the next page, and
3. you can add extra `java` programs if you need.

The following defines keywords and symbols used in this subset of mini-C language:

```
keywords: "int", "print", "var", "func", "if", "then", "else", "while", "void", "begin", "end"
symbols:  "(", ")", ":=", "::", "+", "-", "*", "/", ";", ",", "<", ">", "=", "<>", "<=", ">="
```

The followings describe patterns of number, identifier, newline, and whitespace:

```
number     → [0-9]+("."[0-9]+)?      Positive integer or real values. It can be 12 or 12.34 but not .34 or 12.
identifier → [a-zA-Z][a-zA-Z0-9_]*   C language identifier, except ones that start with '_'
newline    → \n
whitespace → [ \t\r]+
```

Your program should print the following contents in your `Parser` class:

- **tokens** in the form of (**token name , token-attr:lexeme, line number: column number**),
- **successful message** if there is no tokenizing error found: "`success!`", or
- **error message** if there is a tokenizing error:
  "`Error! There is a lexical error at` *line-number*`:`*column number*`.`"

When you run your program with the following "`test4.minc`" and "`test5.minc`" program sources,

- `java Program test4.minc`
- `java Program test5.minc`

where `test4.minc` and `test5.minc` are given as follows:

test4.minc
```
func main::int()
begin
    var a::int;
    a := bcd + 2 * 3;
    b := 1.234
    print a;
end
```

test5.minc
```
mai_n(
begin
       a = a(1-2,3)
    b = 3..;    // number cannot be "3."
begin
```

your program should print the following outputs on console:

```
<FUNC, token-attr:"func", 1:1>
<ID, token-attr:"main", 1:6>
<TYPEOF, token-attr:"::", 1:10>
<INT, token-attr:"int", 1:12>
<LPAREN, token-attr:"(", 1:15>
<RPAREN, token-attr:")", 1:16>
<BEGIN, token-attr:"begin", 2:1>
<VAR, token-attr:"var", 3:5>
<ID, token-attr:"a", 3:9>
<TYPEOF, token-attr:"::", 3:10>
<INT, token-attr:"int", 3:12>
<SEMI, token-attr:";", 3:15>
<ID, token-attr:"a", 4:5>
<ASSIGN, token-attr:":=", 4:7>
<ID, token-attr:"bcd", 4:10>
<OP, token-attr:"+", 4:14>
<NUM, token-attr:"2", 4:16>
<OP, token-attr:"*", 4:18>
<NUM, token-attr:"3", 4:20>
<SEMI, token-attr:";", 4:21>
<ID, token-attr:"b", 5:5>
<ASSIGN, token-attr:":=", 5:7>
<NUM, token-attr:"1.234", 5:10>
<PRINT, token-attr:"print", 6:5>
<ID, token-attr:"a", 6:11>
<SEMI, token-attr:";", 6:12>
<END, token-attr:"end", 7:1>
Success!
```

```
<ID, token-attr:"mai_n", 1:1>
<LPAREN, token-attr:"(", 1:6>
<BEGIN, token-attr:"begin", 2:1>
<ID, token-attr:"a", 3:2>
<RELOP, token-attr:"=", 3:4>
<ID, token-attr:"a", 3:6>
<LPAREN, token-attr:"(", 3:7>
<NUM, token-attr:"1", 3:8>
<OP, token-attr:"-", 3:9>
<NUM, token-attr:"2", 3:10>
<COMMA, token-attr:",", 3:11>
<NUM, token-attr:"3", 3:12>
<RPAREN, token-attr:")", 3:13>
<ID, token-attr:"b", 4:5>
<RELOP, token-attr:"=", 4:7>
Error! There is a lexical error at 4:9.
```

Test cases and points:

8 test cases and their solutions will be provided on the course websites. When grading assignments, 7 additional test cases will be added by the grader. A total of 15 test cases will be used for grading, and 60 points will be assigned to the test cases. Rest 40 points will be assigned based on your implementation: (a) 20 points for **double buffering**, where each **buffer size must be 10**, and (b) 20 points for the use of a **transition diagram** which does not use string comparison except comparing a lexeme and keywords (`int`, `print`, `if`, `else`, etc.). For each test case, you will get

- **1 point** when all **token types** are correctly identified and printed (same as the solution);
- **1 point** when all **lexemes** are correctly identified (same as the solution);
- **1 point** when all line numbers for tokens are correctly determined (same as the solution);
- **1 point** when all column numbers for tokens are correctly determined (same as the solution).

Individual and group submission:

If you want to submit your assignment in a group of 2 members, include your name and your group member's name in the top banner comment of all source files and the **Readme** file. There will be no penalty for group submission.

I encourage you to do the assignment individually. You can improve your programming skills and grasp the whole picture more clearly when you do it by yourself. If you submit your assignment individually, you will get additional points (+3 points). Be aware that some assignments could be difficult, and so you need to select the right strategy between the individual submission and the group submission.

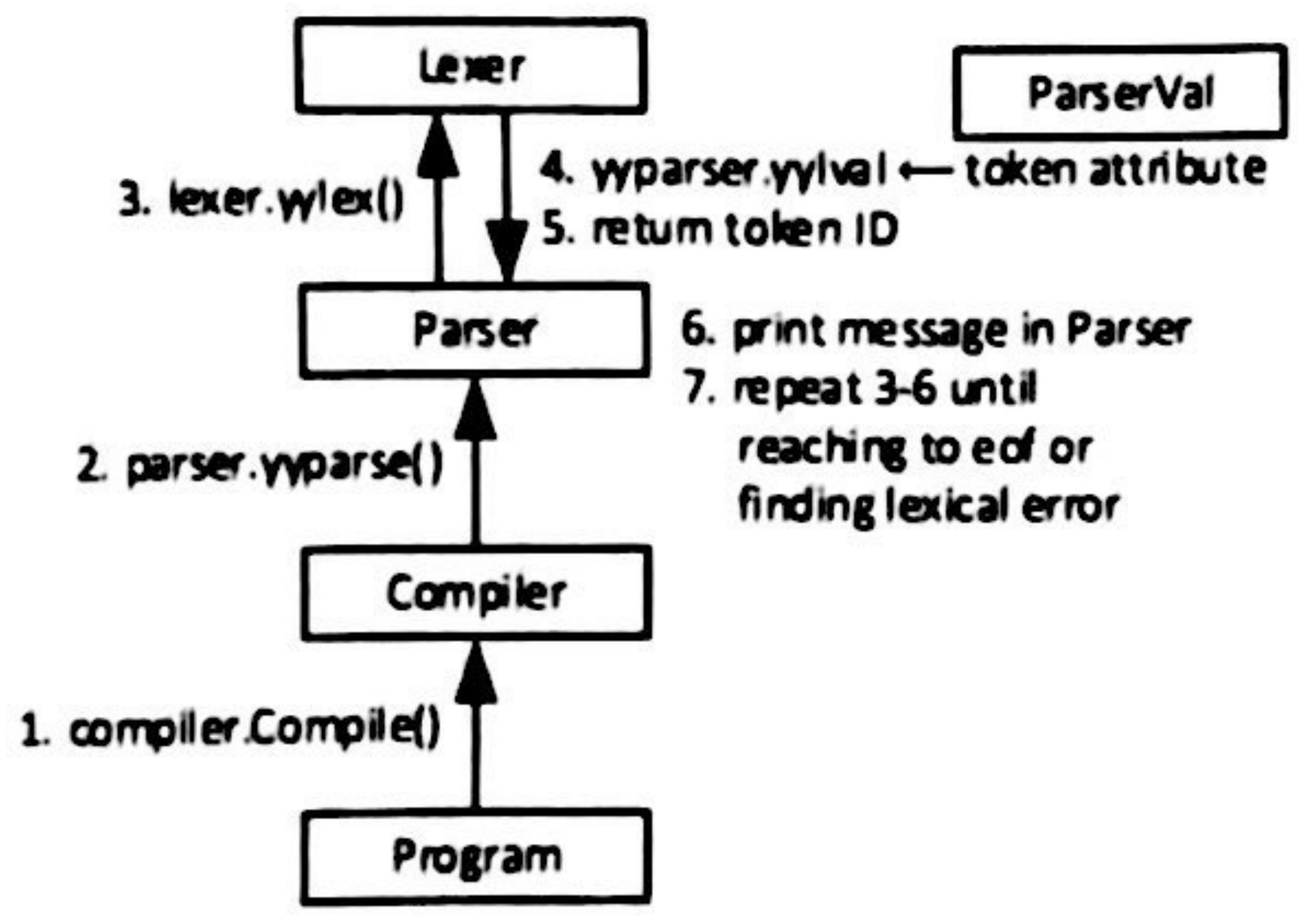
Additionally, the grader will check your source to find i) plagiarism or ii) any use of strange variable names and function names. If you use strange names (such as insulting), you will lose up to 40 points. If plagiarism is identified, it will be reported to the academic integrity committee after double-checking your source files. If the grader cannot compile your program, then you will get only up to 10 points; the grader does not have any responsibility for correcting or debugging your errors.

To submit:

- Submit **one zip file** containing the following files to **Canvas by 11:59:59 PM, Monday, February 5, 2024**.
- **Readme** file describing how to compile your program and other descriptions.
- Your own **`Lexer.java`**, **`Parser.java`**, **`Compiler.java`**, and **other java files** that you use.

7. Do not modify `Program.java` and `ParserVal.java` files; if you submit them, the grader will overwrite them with the startup `Program.java` and `ParserVal.java` files.

Calling structure


Lexer
ParserVal
3. lexer.yylex()
4. yyparser.yylval ← token attribute
5. return token ID
Parser
6. print message in Parser
7. repeat 3-6 until reaching to eof or finding lexical error
2. parser.yyparse()
Compiler
1. compiler.Compile()
Program

When doing this assignment, implementing your solution by following the below sequence will be helpful:

1. Create your own input file (`test0.minc`) that contains only symbols.
2. Implement your `Lexer` that
   a. reads the input file and stores it's all characters into a giant array, and then
   b. determines symbols (only using character comparison).
3. Update your own input file (`test0.minc`) to contain symbols, variable names, and numbers. Then,
   a. draws your own state transition diagram,
   b. updates your `Lexer` that distinguishes symbols, variable names, and numbers
   (only using character comparison)
4. Update your own input file (`test0.minc`) to contain symbols, variable names, numbers, and keywords. Then, update your `Lexer` that distinguishes variable names and keywords (using string comparison).
5. Implement the double buffering mechanism by replacing the giant array with two arrays whose size is 10. Then, test your input file (`test0.minc`).
6. Test the given test `minc` files.
7. Create your own `minc` files and test them to check cases that are not covered in the test `minc` files.