CSE 304 – Compiler Design
Fall, 2023
Assignment 02 – Parsing with PLY

Points: 40
Assigned: Wednesday, 09/13/2023
Due: **Friday, 10/06/2023, at 11:59 PM**

Problem: Parsing `Decaf`

This assignment asks you to build a lexical analyzer and parser for (a subset of) `Decaf`. The description of `Decaf's` syntax is given in its manual (distributed as a PDF with this assignment). Your lexical analyzer and parser will be built using PLY. For this assignment, you are asked to write a "syntax checker" that uses your lexical analyzer and parser. The syntax checker takes a single command-line argument, which is a file name, and determines whether that file contains a syntactically valid `Decaf` program. The syntax checker should only print the string "Yes" to standard output if the program is syntactically valid. If the program is not syntactically valid, then print out an error message. The error message should describe the error and report where it occurs in the input `Decaf` program.

Your syntax checker is only expected to identify and report the *first* syntax error in an input program, even if the program contains several.

Deliverables:

Your syntax checker for `Decaf` should be organized into the following files:

1. **Lexer**: `decaf_lexer.py` – PLY/lex scanner specification file
2. **Parser**: `decaf_parser.py` – PLY/yacc parser specification file
3. **Main**: `decaf_checker.py` – containing the main python function to put together the lexer and parser, take the input from the `Decaf` program file, etc., and perform syntax checking
4. **Documentation**: `README.txt` – which documents the contents of the other files

Input:

Your program should read the name of a file from the command-line. This file will contain the `Decaf` program that your syntax checker must read, parse, and then check for syntax errors. Your program will be run from the command like so:

```
$ python3 decaf_checker.py 1.decaf
```

Output:

The result of the execution of the syntax checker against the `Decaf` program in the input file should be written to standard output.

- "Yes" – if the `Decaf` program is syntactically correct
- An error message describing the first syntax error in the `Decaf` program and where it occurred (line and column number)

Submission Instructions:

1. Submit all of your files in a single compressed archive named: `cse304_a02.zip`
2. Make sure you include your name, netid, and Student ID number as comments at the top of your source code files.
3. Make sure you use any specified, file, function, and class names (and any other names specified in the instructions). Grading may be partially automated and the use of incorrect names will interfere.

This is an independent programming assignment. Any collaboration on coding, with anyone other than your recognized partner, will be considered a violation of academic honesty.

Amendments to (simplifications of) the `Decaf` Specification:

- *Simple string constants* – String constants may themselves not contain the delimiting (**"**) character. Thus, a string constant begins with a (**"**) and ends with a (**"**) with no (**"**) in between.
- *Simple floating point constants* – Floating point constants will only be of the first kind specified in the manual (p. 2). That is: a decimal point with a sequence of one or more digits on either side of it. You are not required to parse the exponent notation for floating point constants.
- *No arrays!!* – The subset of `Decaf` you are expected to recognize does not support the declaration of array-valued fields or variables (p. 3 of the manual), array expressions (array access on p. 6), or array creation (new_array on p. 7).

Assignment Path:

1. Read through the `Decaf` manual and make sure you understand the overall structure of the language.

2. Identify the set of tokens that will be recognized by your lexical analyzer. You may need to revise this set when you work on later steps as well. It may be useful to begin with a slice of `Decaf`, e.g. a (rather useless) subset of the language with no constructor or method definitions, and build a lexical analyzer for the tokens in this subset. Moreover, you will find that definition of certain tokens will be significantly simpler than others; you can specify stubs for the difficult ones at first, then come back to finish the job.

3. In its manual, `Decaf`'s syntax is given in Extended Backus-Naur Form (EBNF). Specifically, the grammar rules in EBNF are of the form:

   `L ::=r1|r2|...|`r_n

   where `L` is a non-terminal symbol (left-hand side of a production) and each r_i is a (possibly empty) right-hand side of a production. In the standard notation of grammars described in class, the right-hand side of a production is a sequence of terminal and non-terminal symbols. In ENBF, the right-hand side may be written using a regular expression-like notation as well. For instance, the following EBNF rule represents a string of one or more `a`'s separated by `b`'s (e.g. `ababa`):

   ```
   S ::= a(ba)*
   ```

   The single rule above can be represented by the set of productions in standard notation below:

   ```
   S → a T
   T → ε
   T → b a T
   ```

   Your task in this assignment is to come up with a **PLY** specification for `Decaf` syntax specified in EBNF. PLY/yacc generates an LALR(1) parser, which means it can only handle a subset (but a large subset) of context-free grammar specifications. Hence, you will need to make sure that the expanded grammar can be handled by PLY. If your specification is ambiguous, make sure that you also specify disambiguating rules unless you are sure that PLY's default behavior is what you want.

   As you construct your PLY/yacc specification, you may need to revise the set of tokens you defined in step 2.

4. Once your token set is nearly completely determined, complete the construction of the lexical analyzer using PLY/lex.

5. Interface your lexical analyzer properly with your parser. Write glue code to take input from a specified file name (command-line argument to your syntax checker), and make your lexical analyzer generate tokens from the contents of that file.

6. Wrap up by adding code to write suitable messages in case of scanning or parsing errors. Make sure the error messages refer to the appropriate line/character position in the input.

Assignment Hints:

- *IMPORTANT: To avoid running into difficulties with conflicts in your grammar*, you should add the grammar rules gradually – as few at a time as possible. Then run through

PLY/yacc to ensure that you do not run into any conflicts. If you do run into conflicts, then the problem is most likely to be in the rules that you just added.

- You may start with only part of the grammar, so long as the portion you take is self-contained, to gain experience with the construction process. For example, you may start with the top-level structure of the language, going through class definitions, field definitions, and method definitions in that order.
- You may also build a syntax checker for a different, small grammar (e.g. the set of valid propositional Boolean formulae) to figure out how the plumbing works, and then build the checker for `Decaf`.
- Make sure you can read and understand the parser.out file produced by PLY/yacc. This file contains the productions, states, and transitions used by the parser. This file will also tell you where the conflicts are, i.e. which states and which productions. From this information, you should be able to make out what the offending grammar rules are and how they should be modified. If, after several attempts, you are unable to resolve the problem contact the instructor or the TA.