Project 2 . Building a Parser

04/09/2020

1 Important information

- **Team:** This is a team project. Each team must have two persons.
- **Grade:** 10% of the final grade of this course.
- **Deliverables:**
  - One design report (as specified in Section 3) including pseudo code and one program.
  - An interview session after your submission. You will be informed of the schedule time.
- **Grading:** Program 50% and design report 50%.
- **Submission:**

  1. By 11:59pm 05/04/2020 , post the following to Blackboard (Please do not wait until last minute. Note that you are allowed to have multiple submissions. ONLY the last one will be graded.)
     (a) The submission is expected to include THREE files: (1) one single compressed file, consisting of all your source code; (2) your design report in PDF format; and (3) a readme in plain text file format with file name "readme.txt". The readme file is about your development environment and how to (compile) and run your program.
     (b) Your name is supposed to appear in every file.

     Your project may not be graded (and thus the grade will be 0) if the requirement above is not followed. Raise any questions about the project and the submission procedure during class.

2 The parser

Write a parser for the following context free grammar based on the tokens (e.g., `id, lparen` etc.) given in project 1.

```
<program> → <stmt_list> $$
<stmt_list> → <stmt> <stmt_list> | ϵ
<stmt> → id assign <expr> | read id | write <expr>
<expr> → <term> <term_tail>
<term_tail> → <add_op> <term> <term_tail> | ϵ
<term> → <factor> <fact_tail>
<fact_tail> → <mult_op> <factor> <fact_tail> | ϵ
<factor> → lparen <expr> rparen | id | number
<add_op> → plus | minus
<mult_op> → times | div
```

Note that `$$` is a pseudo-token (i.e., not a real token in the input program) which is returned by the scan function when it hits the end of the input file and there is no lexical error.

For the parser,

1. its input is a program (a text file).
2. it prints to the console an XML-like tree structure indicating the parse tree of the input program. If the parser meets an unexpected token, your parser should output to the console the following information only and stops: `Error.` Error recovery and handling multiple errors are beyond the scope of this project.
3. you MUST use the recursive descent approach.
4. `scan` function developed for the scanner in project 1 must be used. **Note** the output of `scan` is one token instead of a sequence of tokens.

One way to generate the XML-like tree is as follows. At the beginning of each function for a non-terminal symbol, print to a string buffer a new line containing `[indent]<[Non-Terminal]>`. Before ending the function, print to the buffer a new line of `[indent]</[Non-Terminal]>`. The `indent` is the spaces one should keep before the `Non-Terminal` symbol. When a token is recognized you will print to the buffer `[indent]<[Token-Type]>[Token]</[Token-Type]>`.

Commandline: `parser <input file name>`

Assume the input file has the following program

```
read A
```

Example Output:

```
<Program>
  <stmt_list>
    <stmt>
        <read>
            read
        </read>
        <id>
            A
        </id>
    </stmt>
```

```
    <stmt_list>
    </stmt_list>
  </stmt_list>
</Program>
```

Note. Here, you output both token type and its content. For example, the `id` token type here has a content `A`.

3 Report and Pseudo-code Design

The report consists of four components: 1) Introduction: important ideas and/or data structures you use to build the parser, 2) Pseudo-code: the detailed pseudo-code, 3)Test cases: the test cases with explanation why you select them, and 4) Acknowledgement: acknowledgement of people and their contribution to your project. You may also include any non trivial knowledge you have learned in this project.

Grading considerations/interview questions will be centered around how you print the XML-parse tree correctly and how recursive descent parsing works. One is expected to get a *significant grade deduction if s/he can NOT explain or do NOT understand any part of her/his own report/design*. Therefore, don't write anything into the report and pseudo-code without a reason.

4 Implementation

The programming languages used in this project are restricted to **C/C++, Java, or Python**. As for implementation (program), the minimal expectation is a finished implementation which means the source code is compilable to an executable and the executable behaves correctly at least most of the time.

5 Acknowledgement

Edward Wertz has contributed to the writing of this project.