40-414 - Compiler Design, Fall 1404, Sharif University of Technology

Programming Assignment II (Parser)

Released: Friday, 14/09/1404

Due: Monday, 15/11/1404 at 11:59pm

1. Introduction

In previous assignment, you implemented a scanner. In this assignment you will write a **Predictive Recursive Descent** parser for C-minus by the parsing methods of lecture note 4. Using codes from text books, with a reference to the used book in your program is accepted. In this assignment, you can also use an online toolkit for computing the **First** and **Follow** sets of the non-terminals of the given C-minus grammar at https://mikedevice.github.io/first-follow/, plus a very useful piece of information called **Predict sets** for producing the Parsing Table. However, using codes from the internet and/or other groups/students in this course are **strictly forbidden** and may result in a **fail** grade in the course. Besides, even if you have not implemented the scanner in the previous assignment, you are not permitted to use scanners of other groups. In such a case, you need to implement both scanner and parser for this assignment. If you've announced and worked on previous programming assignment as a pair, you may continue to work on this assignment as **the same pair**, too.

2. Parser Specification

The parser that you implement in this assignment must have the following characteristics:

- The parsing algorithm must be **Predictive Recursive Descent** (i.e., based on the algorithm on pages 76-90 of Lecture Note 4). **Please note that using any other parsing algorithm will not be acceptable and results in a zero mark for this assignment**.
- Parsing is predictive, which means the parser never needs to backtrack.
- Parser works in parallel (i.e., pipeline) with the scanner and other forthcoming modules. In other words, your compiler must perform all tasks in a **single pass**.
- Parser calls `get_next_token` function every time it needs a token, until it receives the last token ($). Note that you need to modify your scanner in such way that it would return the token $, as the last token, when it reaches to the end-of-file of the input.
- Every time that parser calls function `get_next_token`, the current token is replaced by a new token returned by this function. In other words, there is only **one token** accessible to the parser at any stage of parsing.
- Parser recovers from the syntax errors using the **Panic Mode** method discussed on pages 96-98 of Lecture Note 4 (and by using the **follow set** of each non-terminal as its **synchronizing set**).

In this assignment, similar to the previous assignment, the input file is a text file (i.e., named `input.txt`), which includes a C-minus program that is to be scanned and parsed. The parser's outputs include two text files, namely `parse_tree.txt` and `syntax_errors.txt`, which respectively contain the parse tree and possible syntax errors of the input C-minus program. There is no need to print outputs of the scanner in this assignment.

3. C-minus Predictive Grammar

The following grammar is a modified version of the C-minus grammar in [1], where the required conditions to have a **Predictive** parser (i.e., conditions on page 91 of Lecture Note 4) hold. Terminal symbols have a bold typeface in this grammar. **Please note that you must not change or simplify the following grammar in any ways**.

- Program → Declaration-list
- Declaration-list → Declaration Declaration-list
  → EPSILON
- Declaration → Declaration-initial Declaration-prime
- Declaration-initial → Type-specifier **ID**
- Declaration-prime → Fun-declaration-prime
  → Var-declaration-prime
- Var-declaration-prime → **[NUM] ;**
  → **;**
- Fun-declaration-prime → **(** Params **)** Compound-stmt
- Type-specifier → **int**
  → **void**
- Params → **int ID** Param-prime Param-list
  → **void**
- Param-list → **,** Param Param-list
  → EPSILON
- Param → Declaration-initial Param-prime
- Param-prime → **[ ]**
  → EPSILON
- Compound-stmt → **{** Declaration-list Statement-list **}**
- Statement-list → Statement Statement-list
  → EPSILON
- Statement → Expression-stmt
  → Compound-stmt
  → Selection-stmt
  → Iteration-stmt
  → Return-stmt
- Expression-stmt → Expression **;**
  → **break ;**
  → **;**

- Selection-stmt → **if (** Expression **)** Statement Else-stmt

- Else-stmt → **else** Statement
  → EPSILON

- Iteration-stmt → **for (** Expression **;** Expression **;** Expression **)** Compound-stmt

- Return-stmt → **return** Return-stmt-prime

- Return-stmt-prime → Expression **;**
  → **;**

- Expression → Simple-expression-zegond
  → **ID** B

- B → **=** Expression
  → **[** Expression **]** H
  → Simple-expression-prime

- H → **=** Expression
  → G D C

- Simple-expression-zegond → Additive-expression-zegond C

- Simple-expression-prime → Additive-expression-prime C

- C → Relop Additive-expression
  → EPSILON

- Relop → **==**
  → **<**

- Additive-expression → Term D

- Additive-expression-prime → Term-prime D

- Additive-expression-zegond → Term-zegond D

- D → Addop Term D
  → EPSILON

- Addop → **+**
  → **-**

- Term → Signed-factor G

- Term-prime → Factor-prime G

- Term-zegond → Signed-factor-zegond G

- G → ***** Signed-factor G
  → **/** Signed-factor G
  → EPSILON

- Signed-factor → **+** Factor
  → **-** Factor
  → Factor

- Signed-factor-zegond → **+** Factor
  → **-** Factor
  → Factor-zegond

- Factor → **(** Expression **)**
  → **ID** Var-call-prime
  → **NUM**

- Var-call-prime → **(** Args **)**
  → Var-prime

- Var-prime → **[** Expression **]**
  → EPSILON

- Factor-prime → **(** Args **)**
  → EPSILON

- Factor-zegond → **(** Expression **)**
  → **NUM**

- Args → Arg-list
  → EPSILON

- Arg-list → Expression Arg-list-prime

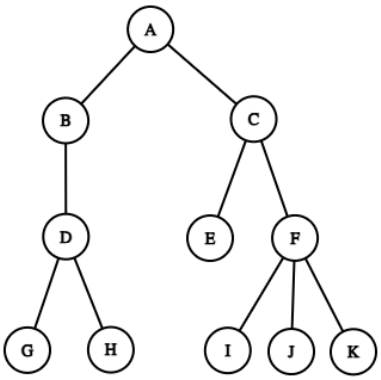
- Arg-list-prime → **,** Expression Arg-list-prime
  → EPSILON

4. Parser Output

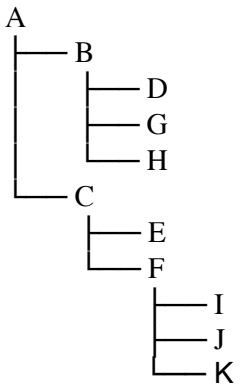
As it was mentioned above, your parser receives a text file named `input.txt` including a C-minus program and outputs the parse tree of the input program in a file named `parse_tree.txt`. Parser also produce a text file called `syntax_errors.txt`, which includes error message regarding possible syntax errors. If there is no syntax error in the input program, a sentence '**No syntax errors found.**' should be written in `syntax_errors.txt`. Therefore, this output file should be created by the parser regardless of whether or not there exists any syntax error.

The parse tree inside `parse_tree.txt` should have the following format:

- Every line includes a node of the parse tree.
- The first line includes the root node, which is the start symbol of the grammar.
- In each line, the depth of the node in that line is shown by a number of tabs before the node's name.



Sample Parse Tree



Sample Output

- The successors of each node from left to right are respectively placed in the subsequent lines.

The following figures show an example a parse tree and the desired format of the output:

5. What to Turn In

Before submitting, please ensure you have done the following:

- It is your responsibility to ensure that the final version you submit does not have any debug print statements.
- You should submit a file named `compiler.py`, which at this stage includes the Python code of your scanner and your **Predictive Recursive Descent** parser. Please write your **full name(s)** and **student number(s)**, and any reference that you may have used, as a comment at the beginning of `compiler.py`.
- Your parser should be the main module of the compiler so that by calling the parser, the compilation process starts, and the parser invokes other modules such as scanner when it is needed.
- The responsibility of showing that you have understood the course topics is on you. Obtuse code will have a negative effect on your grade, so take the extra time to make your code readable.
- If you work in an **announced** pair, please submit **two identical copies** of your assignment (i.e., one copy by each member).
- Your parser will be tested by running the command line `python3 compiler.py` in Ubuntu operating system using Python interpreter version **3.12**. It is a default installation of the interpreter without any added libraries except for **anytree**, which may be needed for creating the parse trees. No other additional Python's library function may be used for this or subsequent programming assignments. Please do make sure that your program is correctly compiled in the mentioned environment and by the given command before submitting your code. It is your responsibility to make sure that your code works properly using mentioned OS and Python interpreter.
- Submitted codes will be tested and graded using several different test cases (i.e., several `input.txt` files) with and without syntax errors. Your program should read `input.txt` from the same working directory as your programs are placed. Please note that in the case of getting a compile or run-time error for a test case, a grade of zero will be assigned to your

parser for that test case (Make sure your code does not return a non-zero exit code in case of success, so that it is not mistaken with an exception or error). Similarly, if the parser cannot produce the expected outputs (i.e., `parse_tree.txt` and `syntax_errors.txt`) for a test case, a grade of zero will be assigned to it for that test case. Therefore, it is recommended that you test your scanner on several different random test cases before submitting your code.

- Your parser will be evaluated by the Quera's Judge System (QJS). These 10 samples will be added to QJS. After the assignment's deadline is passed, new test cases will be substituted for some the sample cases of the Quera and your parser will be judged again.
- The decision about whether the scanner and parser functions to be included in the `compiler.py` or a as separate files such as, say `scanner.py` and `parser.py`, is yours. However, all the required files should be reside in the same directory as `compiler.py`. In other words, I will place all your submitted files in the same plain directory including a test case and execute the `python3 compiler.py` command. You should upload your program files (`compiler.py` and any other text files that your programs may need) to the course page in Quera **before 11:59 PM, Monday, 15/11/1404**.

References

[1] Kenneth C. Louden, Compiler ConstrucIon: Principles and PracIce, 1st Ed., PWS Pub. Co., 1997