40-414 – Compiler Design, Spring of 1403-1404, Sharif University of Technology

Programming Assignment II (Parser)

Released: Sunday, 1404/01/31

Due: Friday, 1404/02/19 at 11:59pm

1 Introduction

In programming assignment 1, you implemented a scanner. In this assignment you will write a **Predictive Top-Down** parser for C-minus, using the **transition diagrams method** on pages 20-31 in lecture note 5. 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 **failure** 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** the **transition diagrams method. Please note that using any other parsing algorithm will not be acceptable and results a zero mark for this assignment**.
- Parsing is predictive, which means the parser never needs to do backtracking.
- 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 invokes 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. (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 Grammar

The following grammar is a modified version of the C-minus grammar in [1], in which the required conditions to have a **Predictive** parser 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**. Neither should you simplify the transition diagrams you build based on this grammar.

1. Program -> DeclarationList
2. DeclarationList -> Declaration DeclarationList | EPSILON
3. Declaration -> DeclarationInitial DeclarationPrime
4. DeclarationInitial -> TypeSpecifier **ID**
5. DeclarationPrime -> FunDeclarationPrime | VarDeclarationPrime
6. VarDeclarationPrime -> **;** | **[NUM] ;**
7. FunDeclarationPrime -> **(** Params **)** CompoundStmt
8. TypeSpecifier -> **int** | **void**
9. Params -> **int ID** ParamPrime ParamList | **void**
10. ParamList -> **,** Param ParamList | EPSILON
11. Param -> DeclarationInitial ParamPrime
12. ParamPrime -> **[ ]** | EPSILON
13. CompoundStmt -> **{** DeclarationList StatementList **}**
14. StatementList -> Statement StatementList | EPSILON
15. Statement -> ExpressionStmt | CompoundStmt | SelectionStmt | IterationStmt | ReturnStmt
16. ExpressionStmt -> Expression **;** | **break ;** | **;**
17. SelectionStmt -> **if (** Expression **)** Statement **else** Statement
18. IterationStmt -> **while (** Expression **)** Statement
19. ReturnStmt -> **return** ReturnStmtPrime
20. ReturnStmtPrime -> **;** | Expression **;**
21. Expression -> SimpleExpressionZegond | **ID** B
22. B -> **=** Expression | **[** Expression **]** H | SimpleExpressionPrime
23. H -> **=** Expression | G D C
24. SimpleExpressionZegond -> AdditiveExpressionZegond C
25. SimpleExpressionPrime -> AdditiveExpressionPrime C
26. C -> Relop AdditiveExpression | EPSILON
27. Relop -> **<** | **==**
28. AdditiveExpression -> Term D
29. AdditiveExpressionPrime -> TermPrime D
30. AdditiveExpressionZegond -> TermZegond D
31. D -> Addop Term D | EPSILON
32. Addop -> **+** | **-**
33. Term -> SignedFactor G
34. TermPrime -> SignedFactorPrime G
35. TermZegond -> SignedFactorZegond G
36. G -> ***** SignedFactor G | EPSILON
37. SignedFactor -> **+** Factor | **-** Factor | Factor
38. SignedFactorPrime -> FactorPrime
39. SignedFactorZegond -> **+** Factor | **-** Factor | FactorZegond
40. Factor -> **(** Expression **)** | **ID** VarCallPrime | **NUM**
41. VarCallPrime -> **(** Args **)** | VarPrime
42. VarPrime -> **[** Expression **]** | EPSILON
43. FactorPrime -> **(** Args **)** | EPSILON
44. FactorZegond -> **(** Expression **)** | **NUM**
45. Args -> ArgList | EPSILON
46. ArgList -> Expression ArgListPrime
47. ArgListPrime -> **,** Expression ArgListPrime | EPSILON

4. Parser Output

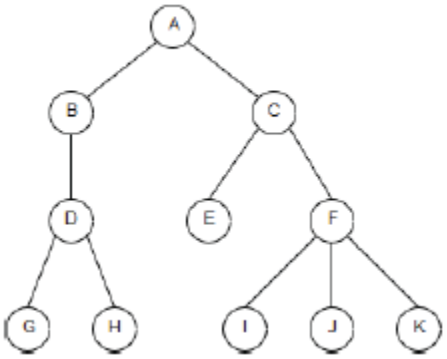
As it was mentioned above, your parser receives a text file named '**input.txt**' including a C-minus program and outputs the parse tree of the input program in a file named '**parse_tree.txt**'. Parser also

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 '**There is no syntax error.**' 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.
- 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.



Sample parse tree

```
A
├── B
|   └── D
|       ├── G
|       └── H
└── C
    ├── E
    └── F
        ├── I
        ├── J
        └── K
```

Sample output

5. Testcases:

- For this phase, you will receive 5 input-output sample files.
- Each group can generate a maximum of 3 completely different input-output sample files and send them to amirrezaazari1381@gmail.com
- Once you submit your test files, they will be added to the Quera page under the section for Phase 2 tests, making them accessible to others.
- Each valid test case earns +0.1 points, with a maximum of +0.3 points available. However, if your test case has an incorrect output, you will receive **-0.1** points.
- Students will report incorrect outputs, and if verified, points will be deducted.
- You only have one chance to submit the files, and you cannot edit them later.
- By working together, you will have many test cases to test your code.
- Duplicate test cases will not be accepted.
- **Deadline: Friday, 1404/02/19 at 08:59pm.**

Email subject format:
Phase2_Tests_Group_(#your_group_number)
Example: Phase2_Tests_Group_15

6. 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 Transition Diagrams** based 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 **only one copy** of your assignment (i.e., one copy by one of the two members who has submitted PA1).
- Your parser will be tested by running the command line '**python3 compiler.py**' in Ubuntu operating system using Python interpreter version **3.8**. 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.
- In a couple of days, you will also receive 5 input-output sample files.
- Your scanner will **not** be evaluated by the Quera's Judge System (QJS).
- 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 resided 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, Friday, 1404/02/19.**

Good Luck!
Reference of C-minus

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