40-414 – Compiler Design, Fall 1401, Sharif University of Technology

Programming Assignment II (Parser)

Released: Sunday, 06/09/1401

Due: Sunday, 27/09/1401 at 11:59pm

1. Introduction

In programming assignment 1, you implemented a scanner. In this assignment you will write a **LR(1)** parser for C-minus, using the **LR(1)** method on pages 42-67 in lecture note 6. Using codes from text books, with a reference to the used book in your program is accepted. In this assignment, you can also use the **bison** toolkit (see section 5) for generating the **SLR(1)** parsing table of the given C-minus grammar. 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** the **LR(1)** method on pages 42-67 in lecture note 6. **Please note that using any other parsing algorithm will not be acceptable and will result in 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 calls the 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 page 67 of Lecture Note 6 (and by using the **follow set** of each non-terminal as its **synchronizing set**). More detailed information is given in section 4.

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.tx**t' 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 table contains a context-free **SLR(1)** grammar for C-minus. 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	return_stmt -> **return ;**
declaration_list -> declaration_list declaration	return_stmt -> **return** expression **;**
declaration_list -> declaration	switch_stmt -> **switch (** expression **) {** case_stmts default_stmt **}**
declaration -> var_declaration	case_stmts -> case_stmts case_stmt
declaration -> fun_declaration	case_stmts -> Epsilon
var_declaration -> type_specifier **ID ;**	case_stmt -> **case NUM :** statement_list
var_declaration -> type_specifier **ID [NUM] ;**	default_stmt -> **default :** statement_list
type_specifier -> **int**	default_stmt -> Epsilon
type_specifier -> **void**	expression -> var **=** expression
fun_declaration -> type_specifier **ID (** params **)** compound_stmt	expression -> simple_expression
params -> param_list	var -> **ID**
params -> **void**	var -> **ID [** expression **]**
param_list -> param_list **,** param	simple_expression -> additive_expression relop additive_expression
param_list -> param	simple_expression -> additive_expression
param -> type_specifier **ID**	relop -> **<**
param -> type_specifier **ID [ ]**	relop -> **==**
compound_stmt -> **{** local_declarations statement_list **}**	additive_expression -> additive_expression addop term
local_declarations -> local_declarations var_declaration	additive_expression -> term
local_declarations -> Epsilon	addop -> **+**
statement-list -> statement_list statement	addop -> **-**
statement-list -> Epsilon	term -> term mulop factor
statement -> expression_stmt	term -> factor
statement -> compound_stmt	mulop -> *****
statement -> selection_stmt	mulop -> **/**
statement -> iteration_stmt	factor -> **(** expression **)**
statement -> return_stmt	factor -> var
statement -> switch_stmt	factor -> call
expression_stmt -> expression **;**	factor -> **NUM**
expression_stmt -> **break ;**	call -> **ID (** args **)**
expression_stmt -> **;**	args -> arg_list
selection_stmt -> **if (** expression **)** statement **endif**	args -> Epsilon
selection_stmt -> **if (** expression **)** statement **else** statement **endif**	arg_list -> arg_list **,** expression
iteration_stmt -> **while (** expression **)** statement	arg_list -> expression

4. Parser Outputs

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'**. Using the **Panic Mode** error recovery method, the parser also produces a text file called '**syntax_errors.txt**', which includes a suitable error message with regard to each detected syntax error.

Whenever you reach an empty cell in the action table. You should do the following things to implement panic mode recovery.

a) Skip the current input symbol and remove as many nonterminal and state pairs from the top of the stack till you reach a state **S** which has a goto to another nonterminal. (you won't have to remove anything from the stack if the initial state has a goto)
b) Discard zero or more input symbols till you reach a symbol that can follow one of the nonterminals in the goto of **S**. Based on the order that you will check the nonterminals you will get different outputs.
   To avoid this check the nonterminals in ascending alphabetical order (a to z).
c) Stack the nonterminal **A** and goto **[S,A]** and continue parsing as normal.

If at any point during your this procedure you had to discard the symbol '**$**' from the input sequence you should halt the parser. There is no need to output a parse tree and you should just report the syntax errors.

The error messages you output should be as following ('lineno' is the line number of 'input.txt' on which the error has occured):

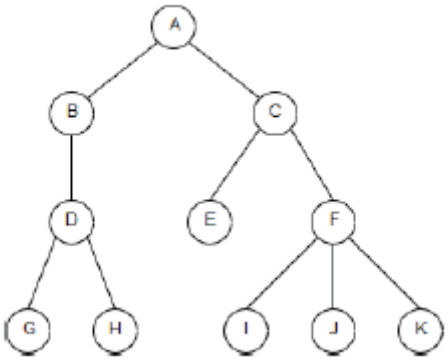
- lineno : syntax error, illegal {terminal in which the error occurs}
- lineno : syntax error, discarded {terminal discarded in step 2} from input
- syntax error, discarded {terminal or nonterminal discarded in step 1} from stack
- lineno : syntax error, missing {non terminal stacked in step 3}
- lineno : syntax error, Unexpected EOF

On the other hand, if there is no syntax error in the input program, the 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. LALR(1) parse table generation

In order to create the LALR(1) parsing table we will use **Bison**. Bison is an open-source application used for creating LALR(1) parsers. **However, we won't be using it to create a parser**. **We will just use it to extract the parse table.**

First of all, install bison version 3.8.2 on your system. If bison is already installed on your system, make sure that it is the same version mentioned above. You can check its version by using the command '**bison --version**' in your terminal.

In order to use bison with the given grammar, we will have to use a specific format. The C-minus grammar has already been transformed to the desired format for you (in a file named 'grammar.y'). We have to run the command '`bison grammar.y --report=all`' in the same directory as the grammar.

After running this command, you will notice two new files that will be created.

1. 'grammar.tab.c' : this is the actual parser produced by bison. We won't be using it at all.
2. 'grammar.out' : this file was created because we asked bison to report everything and it would not have been created otherwise. You can find the LALR states and the action and goto table in this file.

In order to make your job easier, we have provided you with a python program ('parse_table_generator.py') that extracts the parse table and computes the first and follow sets for you. You have to run this program in the same directory as the bison .out file. The output of this program has already been given to you in the project files ('table.json').

However, you will have to add new rules to the grammar in the next phase of the project. So, it is necessary for you to make sure this program works in your system. Practically, this small python program extracts the parse table with the help of regexs. If the bison .out file has a different format in your computer it may not work as expected.

In order to make sure this program works for you, you can run it once and check the output with the 'table.json' file. You probably won't face any problem if you use the same version of bison.

As a word of advice, make sure that your parser only needs the 'table.json' file. This is because when you will change your parser in the next phase, you won't need to change the parser code and you will just have to change the input grammar of bison ('grammar.y').

You may use other applications to extract the LALR(1) parse table but our advice is using bison. If you use another application please make sure it produces a correct LALR(1) parse table.

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 **LR(1)** 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 **3.8**. 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.
- 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.
- Together with this description, you will also receive 10 input-output sample files. You also receive 5 extra input test cases without having access to their expected output.
- Your parser will be evaluated by the Quera's Judge System (QJS). These 15 scases will be added to QJS. After the assignment's deadline is passed, the five test cases plus half of the 10 samples will be used to evaluate your parser.
- 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 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 (https://quera.ir/course/11864/) **before 11:59 PM, Sunday, 27/09/1401**.
- Submissions with more than 100 hours delay will not be graded. Submissions with less than 100 hours delay will be penalized by the following rule:

Penalized mark = M * (100 - D) / 100

Where M = the initial mark of the assignment and D is number of hours passed the deadline.
Submissions with 50 < X <= 100 hours delay will be penalized by P.M. =M * 0.5.

Good Luck!
Gholamreza Ghassem-Sani,
06/09/1401