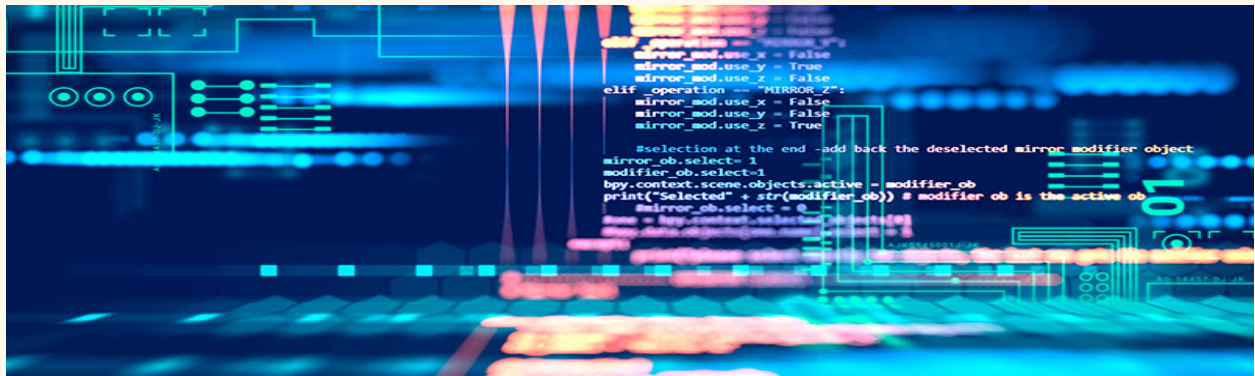


Computer and System Engineering Department

Programming languages and compiler

2022, 2023 semester 1



Phase - 2

Due Date: 4/1/2023

Overview

This phase of the assignment aims to practice techniques for building automatic parser generator tools.

Team

Name	ID
Amr Ahmed Fouad Mohamed Mostafa Al-Dhahabi	18011174
Amr Ayman Ibrahim Momtaz	18011178
Mohannad Tarek Abd El-Khalek Mahmoud	18011889
Momen Ibrahim Fawzy Hassan	18011896

Phase 2: Overview

Reading CFG Input File => Grammer(unorderd_map:(key:string(production name), val:Vector<string>(productions tokens)) =>

Left Recursion and Left Refactoring => LL1 Grammar => First, Follow => Parsing Table Building => parsing and error Handling (using tokenizer)

1-Reading Grammar Input


The module GrammarParser (.h/.cpp) is responsible for reading input grammar for the parser. Simply, when you initialize an instance of it you give it the name of the input file (which should be available in the same directory of executing the code). The constructor will automatically parse all the grammar and it will store the grammar in the map which can be accessed using a getter. So this module has two main functions, one which gets the production map (getter) and the other function just returns the name of the starting symbol of the given grammar (the first production of the grammar). This module handles all spaces and line breaks and (') which are used for non-terminals.

Assumptions:

No assumptions were taken.

Data Structures and modules:

We use unordered_map<string, vector<vector<string>>> to store the productions such that. The key in the map represents the production name and the value will be a vector of vectors of string. We used a vector of vector such that every two production rules separated with | operator will be in two different vectors. We used this to distinguish between statements which are unioned together.



2-Left Recursion and Left Refactoring (Bonus)

In this part we take the input CFG grammar and convert it into an LL(1) grammar by eliminating the left recursion and left factoring.

We start by eliminating left recursion where we loop over all productions after sorting them in any arbitrary way then substituting the non terminals in the current production by the found non terminals in the previous productions then checking for and eliminating any direct left recursions that appear. Afterwards we eliminate left factoring by looping over all the productions and checking if at least two productions share the same starting terminal / non terminal then taking it as a common factor and adding the rest into a new terminal while adding or between the terminals / non terminals and leaving the ones that didn't have the same factor untouched in the original production.

Assumptions:

(ϵ) is written as "Epsilon" in the grammar file as written in the Java CFG provided in the project phase PDF.

Data Structures:

Only data structure used is an unordered map called **productions** which carries the LHS of the production and a vector of vectors for the RHS of the productions.

Main Methods:

- **canBeFactored(string)**: method that checks whether the given production has left factors that can be eliminated.
- **canBeLeftRecursed(string)**: method that checks whether the given production has direct left recursion that can be eliminated.

- **substitute(string1, string2)**: method that substitutes the non terminal string2 if it exists as the first element in the production of the non terminal string1.
- **eliminate_left_recursion(string)**: method that eliminates direct left recursion in the given production.
- **eliminate_left_factoring(string)**: method that eliminates left factoring in the given production.
- **eliminate_indirect_left_recursion()**: method that eliminates all left recursions in the given CFG grammar.

3-Get the first and the follow

In this part we take the LL(1) grammar and compute the First and Follow for each non terminal. Each element in the First set is associated with its corresponding production.

Computing the First set of each non terminal for example First(S) was done w.r.t three rules :

1. If we encounter Epsilon or a terminal token, then add it to the First(S) and proceed to the next production.
2. If we encounter a non terminal and its First set does not contain Epsilon, then add this set to the First(S) and proceed to the next production.
3. If we encounter a non terminal and its First set contains Epsilon, then add this set except Epsilon to the First(S) and continue. If we reached the last token in the production and still have Epsilon, then add it to the First(S).

In case the non terminal is the token of interest, then proceed to the next production.

In case the non terminal is not yet computed, skip it and continue looping.

This method does not use recursion.



Computing the Follow set of each non terminal for example Follow(S) was done w.r.t Four rules :

1. If S is the last token in a production, then add Follow(head) to the Follow(S).
2. If S has a following terminal, then add it to the Follow(S) and proceed to the next production.
3. If S has a following non terminal and its First set does not contain Epsilon, then add its First set to the Follow(S) and proceed to the next production.
4. If S has a following non terminal and its First set contains Epsilon, then add its First set except Epsilon to the Follow(S) and continue.

We add the "\$" symbol to the starting non terminal.

In case the non terminal is the token of interest, then proceed to the next production.

In case the non terminal is not yet computed, then recurs on it.

This method uses recursion.


Assumptions:

The only assumption is that the given grammar is LL(1) which means no cycles could be introduced.

Data Structures and modules:

We have only one module here and 2 main data structures. One for (**Follow**) and another one for the First set associated with the corresponding production (**associated_prod**).

We have other helper data structures like (**indices**) and (**done**). Both are used in computing the First set (as we don't use recursion here). We have (**done**) storing the computed non terminals so far. We have (**indices**) storing the skipped non terminals and the corresponding last index (i, j) checked.



4-Building the parsing table

The parser needs two things to start the parsing operation: the Lexical analyzer to get the next tokens when we need and the parsing table.

As we made the Lexical analyzer in phase 1 of that project then now we need to build the parsing table.

So, we get the first and the follow of the input grammar to build the parsing table that is used in the parser with the lexical analyzer as we mentioned above.

Building the parsing table:

To build the parsing table we put the next production of each current production in the places of the given 'first' of the current production and

if that 'first' contains "Epsilon" we will add the 'follow' of the current production and make the next production at these places equals "Epsilon"

If not we will add the 'follow' of the current production and make the next production at these places equals "sync".

Assumptions:

When we build the parsing table if the grammar is ambiguous the program will stop and print an error message of that.

In phase 2 the '=' terminal named 'assign' but in that phase its name was '=' so, we modified the current input grammar by replacing each '=' by 'assign' which is the right terminal name.

Data Structures and modules:

We used an unordered map called a **parsing table** which carries the names of the production and an unordered map of the input terminal and the corresponding next production.

```
unordered_map<string, unordered_map<string, vector<string>>> parsing table;
```

Output (Example):

We get the parsing table of the given grammar as follow: [file](#)

```

----- Parsing Table -----
EXPRESSION_factor_3_factor_9_factor_14:
relop: relop SIMPLE_EXPRESSION
;: Epsilon
): Epsilon

EXPRESSION_factor_4:
relop: TERM_recur_0 EXPRESSION_factor_4_factor_12
addop: TERM_recur_0 EXPRESSION_factor_4_factor_12
;: sync
): sync
mulop: TERM_recur_0 EXPRESSION_factor_4_factor_12

.
.
.
.
.

METHOD_BODY:
id: id assign EXPRESSION ; STATEMENT_LIST_recur_2
if: if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST_recur_2
float: float id ; STATEMENT_LIST_recur_2
while: while ( EXPRESSION ) { STATEMENT } STATEMENT_LIST_recur_2
$: sync
int: int id ; STATEMENT_LIST_recur_2

EXPRESSION:
-: - EXPRESSION_factor_7
+: + EXPRESSION_factor_6
(: ( EXPRESSION_factor_5
num: num EXPRESSION_factor_4
;: sync
): sync
id: id EXPRESSION_factor_3

----- End of Parsing Table -----

```

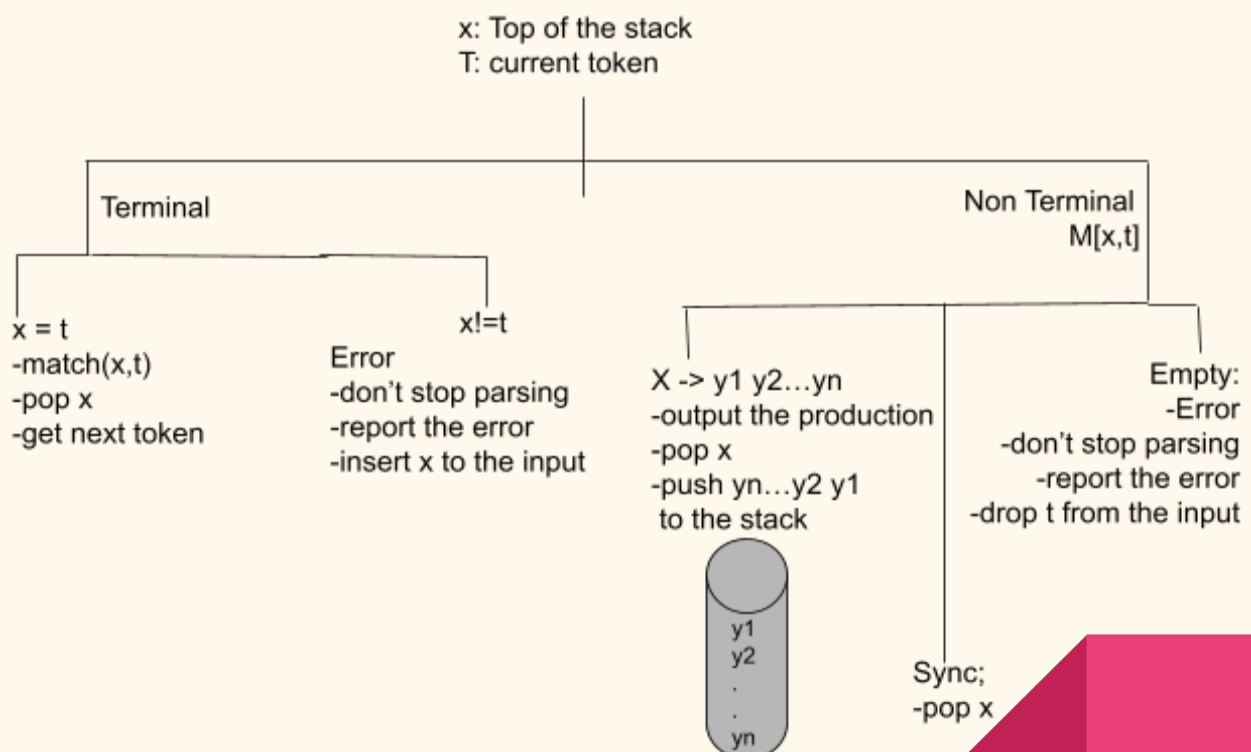
5-Parsing and Error handling

In this part the parsing starts to work using the parsing table and the Lexical analyzer to get the input file in only **one pass**. Using stack as:

The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.

There are four possible parser actions:

1. If X and a are \$ parser halts (successful completion)
2. If X and a are the same terminal symbol (different from \$) parser pops X from the stack, and moves the next symbol in the input buffer.
3. If X is a non-terminal parser look at the parsing table entry $M[X,a]$. If $M[X,a]$ holds a production rule $XY_1Y_2\dots Y_k$, it pops X from the stack and pushes Y_k, Y_{k-1}, \dots, Y_1 into the stack. The parser also outputs the production rule $XY_1Y_2 \dots Y_k$ to represent a step of the derivation.
4. none of the above errors – all empty entries in the parsing table are errors. – If X is a terminal symbol different from a, this is also an error case.



Data Structures and modules:

Stack:

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol \$.
- initially the stack contains only the symbol \$ and the starting symbol S.
\$S initial stack
- when the stack is emptied (ie. only \$ left in the stack), the parsing is completed.

The output of our program

If the input file is: [file](#)

```
Test Program
int sum , count , pass , mnt; while (pass !=
10)
{
    pass = pass + 1 ;
}
```

The leftmost derivation: [file](#)

```
----- The Leftmost Derivation Sentential -----
METHOD_BODY
int id ; STATEMENT_LIST_recur_2
int id ; STATEMENT STATEMENT_LIST_recur_2
int id ; id assign EXPRESSION ; STATEMENT_LIST_recur_2
int id ; id assign id EXPRESSION_factor_3 ; STATEMENT_LIST_recur_2
int id ; id assign id sync ; STATEMENT_LIST_recur_2
int id ; id assign id sync ; STATEMENT STATEMENT_LIST_recur_2
int id ; id assign id sync ; while ( EXPRESSION ) { STATEMENT } STATEMENT_LIST_recur_2
.
.
.
int id ; id assign id sync ; while ( id Epsilon Epsilon relop num Epsilon Epsilon ) {
id assign id Epsilon addop num Epsilon Epsilon Epsilon ; } Epsilon
```

If the input file is: [file](#)

```
int x;
x = 5;
if (x > 2)
{
    x = 0;
}
```

The leftmost derivation: [file](#)

```
----- The Leftmost Derivation Sentential -----
METHOD_BODY
int id ; STATEMENT_LIST_recur_2
int id ; STATEMENT STATEMENT_LIST_recur_2
int id ; id assign EXPRESSION ; STATEMENT_LIST_recur_2
int id ; id assign num EXPRESSION_factor_4 ; STATEMENT_LIST_recur_2
int id ; id assign num sync ; STATEMENT_LIST_recur_2
int id ; id assign num sync ; STATEMENT STATEMENT_LIST_recur_2
int id ; id assign num sync ; if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST_recur_2
.
.
.
int id ; id assign num sync ; if ( id Epsilon Epsilon relop num Epsilon Epsilon ) { id
assign num sync ; } else { sync } Epsilon
```

Appendix:

The used codes and executables can be found here:

https://drive.google.com/drive/folders/1Dkub9cBQD6RO_sefco4nJSILzloRYtjz?usp=sharing