Syntax Directed Translation Scheme

Compiler Project

Phase 2

—

Names	**ID**
• Afnan Mousa	15
• Enas Morsy	20
• Sara Mohammad	31
• Shimaa Kamal	34

Main Algorithm:

- ❏ Read from the “ **grammar.txt** “ file.
- ❏ Fix the **Left factor**.
- ❏ Fix the **Left recursion**.
- ❏ Calculate the **First set** of each non terminal and store the needed data to calculate the **Follow set**.
- ❏ Calculate the **Follow set** of each nonterminal.
- ❏ Generate the **parsing table**.
- ❏ Get the next token from the **lexical** phase , parse it and generate the output.
- ❏ Repeat the previous step until it gets the “ **$** “ sign as the end of the input tokens.

Classes:

- ❏ Main class
- ❏ Parse_grammer class
- ❏ Left_factor class
- ❏ Generat_follow class
- ❏ Generate_table class
- ❏ Check class

Reading Grammar

Functions:

- ❏ **readFile**(char *filename**)** : main class.

Algorithm:

- ★ Read the file char by char from the **end**.
- ★ Append the new character to the **start** of the line string **only if** it isn’t a **‘ \n ’** char.
- ★ If the new character is **‘ \n ‘**, check the previous character.
- ★ If the previous character is **‘ # ‘** , Fix the **left factor** and the **left recursion** and empty the **line** string .

- ★ Otherwise it’s a **multi line**, ignore the ‘ \n ’ char and continue reading characters and appending them to the start of the same line.

Assumptions:

- ★ Reading the grammar text file only **once** from the end to calculate the First set and store what is needed to calculate the Follow set later.

Left factoring

Data structures:

- ❏ **vector <string> productions**, holds all the productions on a line.
- ❏ **vector < vector<int> > common,** holds the productions that have a common prefix together.
- ❏ **map <string, string> new_production,** stores the new productions derived from left factoring.

Functions:

- ❏ **Left_factoring () :** Left_factor class
  This function has the main algorithm for left factoring and it calls almost all other functions in this class.
  Its algorithm will be in the Algorithm section below.

- ❏ **Common_prefix (**productions**) :** Left_factor class
  This function takes all the productions and starts to split them into groups. Each group has the same common prefix. For example if we have this production:
  A → ad | a | ab | abc | b
  Then it will split the productions to 2 groups:
  G1 = (**a**d | **a** | **a**b | **a**bc)
  G2 = (b)

- ❏ **Longest_common_prefix (**group of common productions**) :** Left_factor class
  This function takes a group of common prefix productions as a parameter and starts to iterate on them parallely to get the longest common prefix string

of them.

- **Check_left_fact (**common productions groups**) :** Left_factor class
  This function takes all the common production groups and starts to iterate on them to check if the size of any group is more than one, this means there is a left factoring needed to be fixed. And of course if each group has only a 1 production, this means there is no left factoring in this production.

Algorithm:

- The main algorithm for left factoring is almost done on the **Left_factoring()** function. Which takes the whole line as a parameter and starts to split it into a non-terminal part and a production part.
- Then it again splits the production part into small productions with " | " operator.
- Then it splits these productions into groups where each group has the same prefix by calling the **Common_prefix (**productions**)** function and passes all the small productions to it.
- After returning from it, it checks if there is a need for left factoring or not by calling **Check_left_fact ()** function, and if there is no need for left factoring, return from the whole algorithm.
- Else, it starts to iterate on each group of a common prefix and get its prefix string by calling **Longest_common_prefix().**
- Then it starts to reconstruct the main line by concatenating the prefix and the name of the new production with it.
  For example: **Expression = prefix Expression_1**. (1 refers to dash sign in the lectures).
- Then it starts to construct the new productions that don't have left factoring on it, by iterating on them to take the remaining string of each small production after removing the prefix from it, ans concatenate all of them together using " | " operator.
- After finishing each iteration, it inserts the new production into a map that holds the constructing productions.
  And after finishing the whole iterations, it inserts the main line into the map which will be returned and passed to the main class to add these productions to the main productions.
- The last important task for this algorithm, that we again iterate on those constructing productions to check if any one of them also has a left factoring, and if so, the function recursively calls itself to fix left factoring.
  For example, if we have this production:
  Here A' also has a left factoring, so it recursively calls the function so that the

final result will be:
A → **a**d | **a** | **a**b | **a**bc | b

Then after the first call it will be:
A → aA' | b
A' → d | \L | **b** | **b**c

Here A' also has a left factoring, so it recursively calls the function so that the final result will be:
A → aA' | b
A' → d | \L | bA"
A" → \L | c
As will be shown in the sample runs.

Sample Runs:

★ **1st lecture example:**

grammar - Notepad

File Edit Format View Help

```
# A = a b B | a B | c d g | c d e B | c d f B
```

Output after fixing left-factoring:

```
# A = a A1 | c d A2
# A1 = b B | B
# A2 = g | e B | f B
```

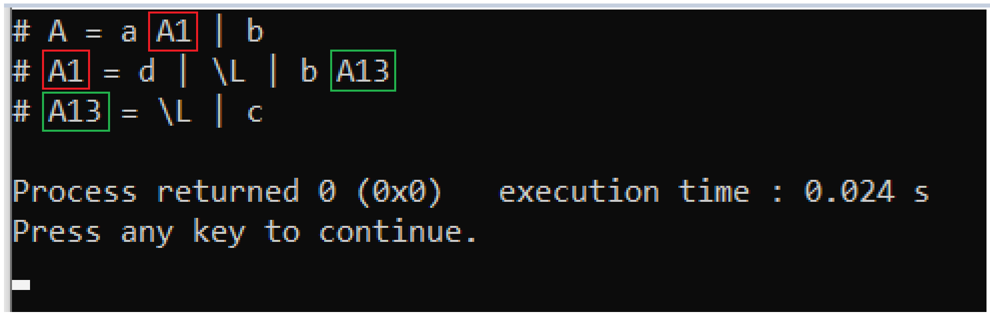
★ **2nd lecture example:**

grammar - Notepad

File Edit Format View Help

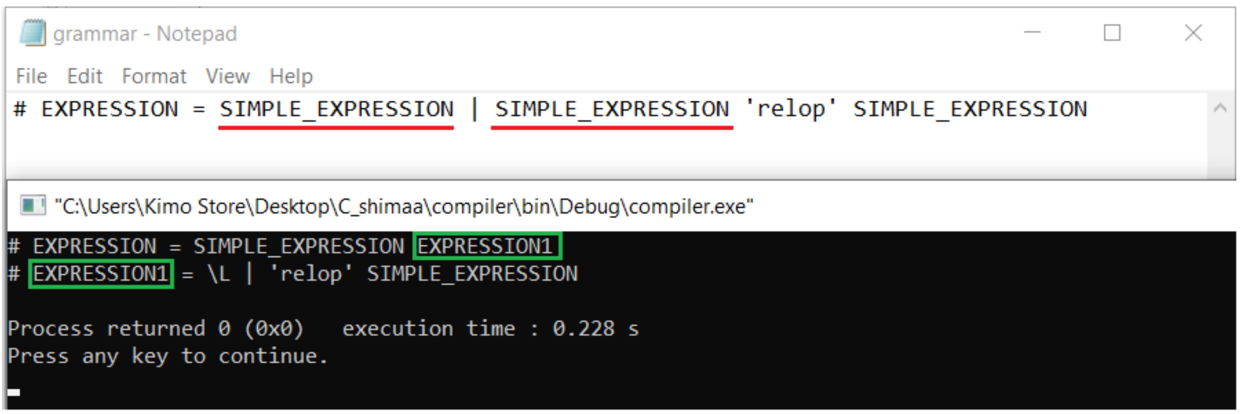
```
# A = a d | a | a b | a b c | b
```

Output after fixing left-factoring:



```
# A = a A1 | b
# A1 = d | \L | b A13
# A13 = \L | c

Process returned 0 (0x0)   execution time : 0.024 s
Press any key to continue.
```

★ **Provided grammar after applying left-factoring for EXPRESSION production:**



```
grammar - Notepad
File  Edit  Format  View  Help
# EXPRESSION = SIMPLE_EXPRESSION | SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION

"C:\Users\Kimo Store\Desktop\C_shimaa\compiler\bin\Debug\compiler.exe"
# EXPRESSION = SIMPLE_EXPRESSION EXPRESSION1
# EXPRESSION1 = \L | 'relop' SIMPLE_EXPRESSION

Process returned 0 (0x0)   execution time : 0.228 s
Press any key to continue.
```

Left recursion

Data structures:

❏ **Vector < string > result**: as each production is changed to multiple productions after fixing the left recursion.

Functions:

❏ vector<string> **Left_Recursion(**string line**)** : main class.
   - ➢ Erase the # sign from the start of the production.
   - ➢ Split the production using the " = " sign to get 2 parts: the first one is the non terminal and the second part is the derivations.
   - ➢ Split the second part using the " | " sign.

- ➢ Loop on the derivations , split each derivation using the space sign to get the first part and check if it equals the non terminal part, store the index of the part of the production which has left recursion and break from the loop.
- ➢ If there is no left recursion the index is still -1 , push the production in the result and return it.
- ➢ Otherwise the new non terminal has the same name of the old terminal plus the “ ~ “ sign.
- ➢ Get the part of the derivation without the first part that caused the left recursion.
- ➢ Declare the new production as 2 derivations and between them “ | “ sign :

  - the part followed by the new non terminal.

  - epsilon.

- ➢ Declare the old production as all the derivations that hasn’t left recursion separated by “ | “ sign . each derivation is followed by the new non terminal.
- ➢ Push the two productions in the result vector and return it.

❏ string fix_left_recursion (string line) : main class.
- ➢ Call the left_recursion function to get the result vector.
- ➢ Loop on the result vector and call the extract_from_line() using each production.

Algorithm:

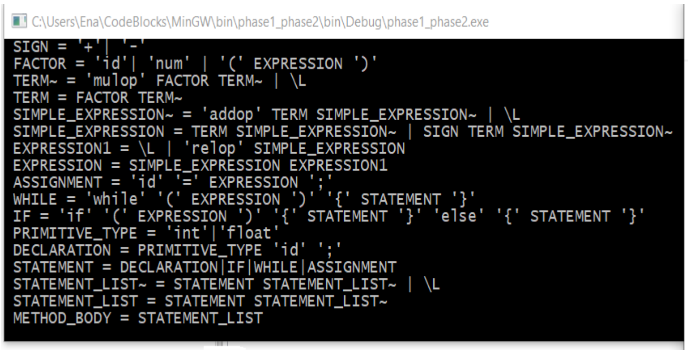
★ Call the fix_left_resursion() function.

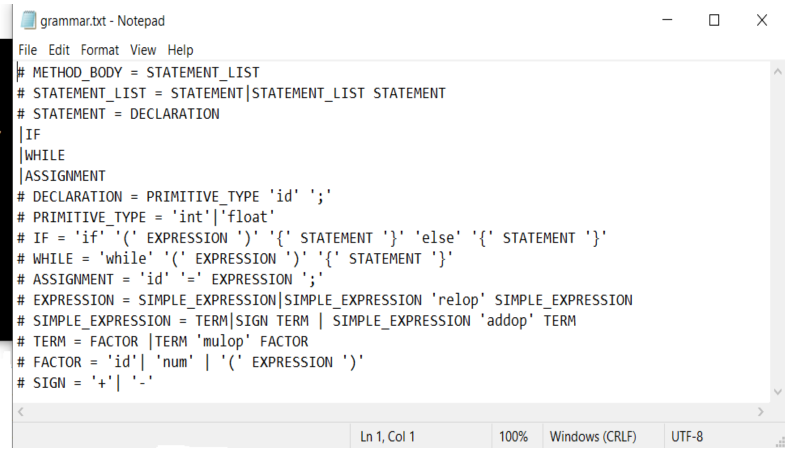
Assumptions:

★ Only intermediate left recursion is fixed.

★ Left factor is called first so each production has only one derivation that has left recursion.

Sample Runs:





★ TERM , SIMPLE_EXPRESSION and STATEMENT_LIST have left_recursion should be fixed.

Generate First set

Data structures:

❏ **map<string, vector<pair<string, string>>> First** : each non terminal has a vector of pairs to express its First set , pair consists of 2 strings the first one is the terminal and the second one is the derivation caused to have this terminal.

Functions:

❏ String **extract_from_line(**string line**)** : parse_grammar class.
- ➢ After splitting the production , Loop on the derivations within production.
- ➢ Initialize vector temp to store the first set of this non terminal.
- ➢ Split each derivation using space.
- ➢ Check the first part of each derivation :
  - If it is terminal (starts with ‘) : erase the single quote and push it in the temp vector.
  - If it’s epsilon : push it in the temp vector.
  - Otherwise it’s non terminal : push its first set in the temp vector and while the non terminal has epsilon as its first, push the first of the next non terminal and so on. push the epsilon in the

temp vetcor only if all the non terminals have epsilon in their first set.

➢ Push in the First map the non terminal as key and the temp vector as value.
➢ Returns the non terminal to be stored as start_grammar.

Algorithm:

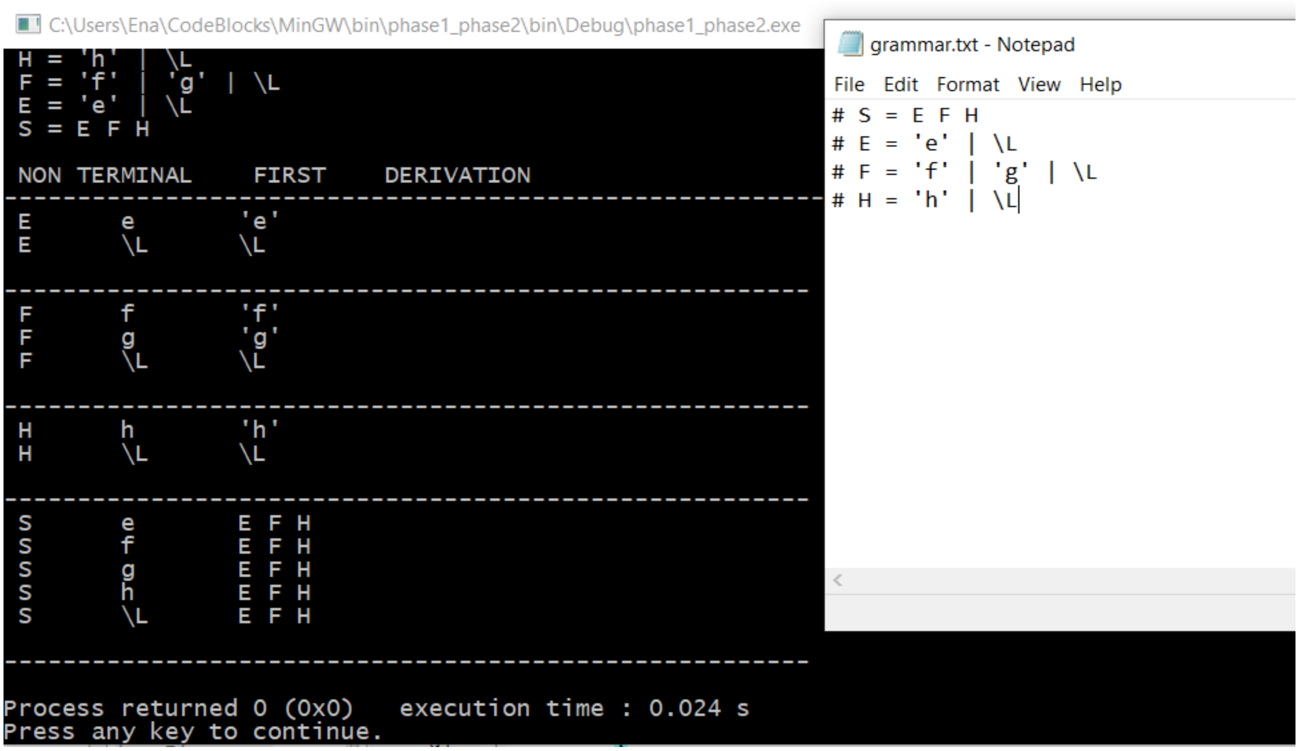
★ Call **extract_from_line ()** to insert all the First set of the non terminal to be used later in generating the parsing table.

Assumptions:

★ No assumptions.

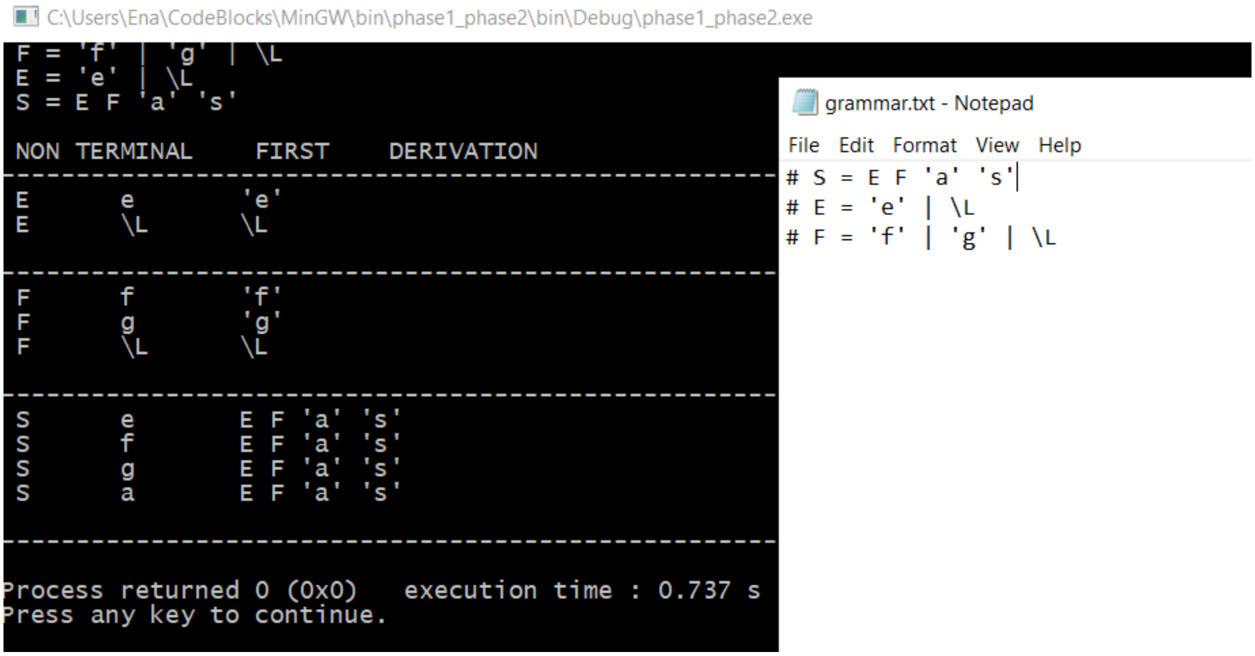
Sample Runs:

★ **Special case** : in production S there exists E,F and H all have epsilon.



★ **Special case** : in production S there exists E and F have epsilon then ‘a’.

" don’t add epsilon “.



★ **The grammar provided in pdf** :

Select C:\Users\Ena\CodeBlocks\MinGW\bin\phase1_phase2\bin\Debug\phase1_phase2.exe

```
NON TERMINAL     FIRST     DERIVATION
---------------------------------------------------------------------
ASSIGNMENT      id       'id' '=' EXPRESSION ';'

-------------------------------------------------------
DECLARATION     int      PRIMITIVE_TYPE 'id' ';'
DECLARATION     float    PRIMITIVE_TYPE 'id' ';'

-------------------------------------------------------
EXPRESSION      id       SIMPLE_EXPRESSION EXPRESSION1
EXPRESSION      num      SIMPLE_EXPRESSION EXPRESSION1
EXPRESSION      (        SIMPLE_EXPRESSION EXPRESSION1
EXPRESSION      +        SIMPLE_EXPRESSION EXPRESSION1
EXPRESSION      -        SIMPLE_EXPRESSION EXPRESSION1

-------------------------------------------------------
EXPRESSION1     \L       \L
EXPRESSION1     relop    'relop' SIMPLE_EXPRESSION

-------------------------------------------------------
FACTOR id       'id'
FACTOR num      'num'
FACTOR (        '(' EXPRESSION ')'

-------------------------------------------------------
IF     if       'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'

-------------------------------------------------------
METHOD_BODY     int      STATEMENT_LIST
METHOD_BODY     float    STATEMENT_LIST
METHOD_BODY     if       STATEMENT_LIST
METHOD_BODY     while    STATEMENT_LIST
METHOD_BODY     id       STATEMENT_LIST
```

```
STATEMENT_LIST~         int     STATEMENT STATEMENT_LIST~
STATEMENT_LIST~         float   STATEMENT STATEMENT_LIST~
STATEMENT_LIST~         if      STATEMENT STATEMENT_LIST~
STATEMENT_LIST~         while   STATEMENT STATEMENT_LIST~
STATEMENT_LIST~         id      STATEMENT STATEMENT_LIST~
STATEMENT_LIST~         \L      \L

-------------------------------------------------------
TERM    id      FACTOR TERM~
TERM    num     FACTOR TERM~
TERM    (       FACTOR TERM~

-------------------------------------------------------
TERM~   mulop   'mulop' FACTOR TERM~
TERM~   \L      \L

-------------------------------------------------------
WHILE   while   'while' '(' EXPRESSION ')' '{' STATEMENT '}'

-------------------------------------------------------
```

```
-------------------------------------------------------
PRIMITIVE_TYPE int      'int'
PRIMITIVE_TYPE float    'float'

-------------------------------------------------------
SIGN    +       '+'
SIGN    -       '-'

-------------------------------------------------------
SIMPLE_EXPRESSION       id      TERM SIMPLE_EXPRESSION~
SIMPLE_EXPRESSION       num     TERM SIMPLE_EXPRESSION~
SIMPLE_EXPRESSION       (       TERM SIMPLE_EXPRESSION~
SIMPLE_EXPRESSION       +       SIGN TERM SIMPLE_EXPRESSION~
SIMPLE_EXPRESSION       -       SIGN TERM SIMPLE_EXPRESSION~

-------------------------------------------------------
SIMPLE_EXPRESSION~      addop   'addop' TERM SIMPLE_EXPRESSION~
SIMPLE_EXPRESSION~      \L      \L

-------------------------------------------------------
STATEMENT       int     DECLARATION
STATEMENT       float   DECLARATION
STATEMENT       if      IF
STATEMENT       while   WHILE
STATEMENT       id      ASSIGNMENT

-------------------------------------------------------
STATEMENT_LIST int      STATEMENT STATEMENT_LIST~
STATEMENT_LIST float    STATEMENT STATEMENT_LIST~
STATEMENT_LIST if       STATEMENT STATEMENT_LIST~
STATEMENT_LIST while    STATEMENT STATEMENT_LIST~
STATEMENT_LIST id       STATEMENT STATEMENT_LIST~

-------------------------------------------------------
```

Generate Follow set

Data structures:

- **Map <string, vector<string>> follow,** that maps each nonterminal to its followers.
- **Vector <string> ordered_follow,** that stores all the nonterminals with their appearing order in the grammar file to calculate the follow set.
- **Map helper_Follow,** that maps each nonterminal to:
    1. The LHS of each production that it appears on.
    2. The nonterminals which follow it.
    3. The LHS of the production if this terminal is the most right on the production.

Functions:

- **prepare_follow (**LHS, production**):** parse_grammar Class.
  This function is called for each small production (after splitting the main one by “ | “ operator). Then it iterates on each nonterminal on this production and maps it to its follower and its LHS nonterminal that will be needed for generating the follow set.
  And maps it to LHS nonterminals also if this nonterminal is the most right.

- **get_follow()**: generate_follow Class.
  This function implements the main algorithm for generating the followers for all the nonterminals with its special cases. It will be explained in detail in the Algorithm section below.

Algorithm:

- For generating the follow set, **get_follow()** starts to iterate on the nonterminal with its appearing order on the grammar.txt.
- Initially, for the first nonterminal it pushes the dollar sign “$” to its follow vector.
- Then it iterates on the followers of each nonterminal which are stored on the **helper_Follow** map. And checks if this follower is a terminal, then pushes it directly to the follow vector.

Else, it pushes all the first vector elements of this nonterminal to the current nonterminal follow vector except the epsilon.

★ If the nonterminal follower contains the epsilon on its first vector, then it pushes the follower of the LHS nonterminal of this production to the current nonterminal follow vector.

★ Finally, using help of the **helper_Follow** map, we iterate on a vector that holds the the LHS of the productions which the current nonterminal appears on them as a most right noterminal, and push the follow of these LHS of the follow vector of the current nonterminal.

Sample Runs:

★ **Lecture example:**

```
grammar - Notepad
File  Edit  Format  View  Help
# E = T E'
# E' = '+' T E' | \L
# T = F T'
# T' = '*' F T' | \L
# F = '(' E ')' | 'id'
```

Follow set:

```
Follow (E)  = {  $,  )  }
Follow (E')  = {  $,  )  }
Follow (F)  = {  $,  ),  *,  +  }
Follow (T)  = {  $,  ),  +  }
Follow (T')  = {  $,  ),  +  }

Process returned 0 (0x0)   execution time : 0.178 s
Press any key to continue.
```

★ **Follow set for the grammar provided in the lab**

```
Follow (ASSIGNMENT)  = {  $,  float,  id,  if,  int,  while,  }  }
Follow (DECLARATION)  = {  $,  float,  id,  if,  int,  while,  }  }
Follow (EXPRESSION)  = {  ),  ;  }
Follow (EXPRESSION1)  = {  ),  ;  }
Follow (FACTOR)  = {  ),  ;,  addop,  mulop,  relop  }
Follow (IF)  = {  $,  float,  id,  if,  int,  while,  }  }
Follow (METHOD_BODY)  = {  $  }
Follow (PRIMITIVE_TYPE)  = {  id  }
Follow (SIGN)  = {  (,  id,  num  }
Follow (SIMPLE_EXPRESSION)  = {  ),  ;,  relop  }
Follow (SIMPLE_EXPRESSION~)  = {  ),  ;,  relop  }
Follow (STATEMENT)  = {  $,  float,  id,  if,  int,  while,  }  }
Follow (STATEMENT_LIST)  = {  $  }
Follow (STATEMENT_LIST~)  = {  $  }
Follow (TERM)  = {  ),  ;,  addop,  relop  }
Follow (TERM~)  = {  ),  ;,  addop,  relop  }
Follow (WHILE)  = {  $,  float,  id,  if,  int,  while,  }  }
```

Generate Parsing Table

Data structures:

- map<string,map<string,string>> **Parsing_Table** : the key of **outer** map is the **non terminal** and the **inner** map its key is the **terminal** and its value is the **entry** of the table .

Functions:

- **get_Parsing_Table(First, Follow)** : generate_table class.
  - Declare map <string,string> temp.
  - Loop on the First map.
  - For each non terminal Loop on its vector stored in First map.
  - Check if the current first isn't epsilon , check if already the temp has this first (it means 2 entries for the same non terminal and terminal so print message " NOT LL(1) GRAMMAR !! " and exit the program) otherwise insert this first and its derivation in the temp map.

- ➢ Otherwise if it's epsilon, set flag equals to true to be used later.
- ➢ Loop on the Follow map.
- ➢ If the flag is set to have epsilon in its First set , insert all the Follow set in the temp map with derivation " \L " if it does not exist and if it exists, the same message should be printed.
- ➢ If it doesn't contain epsilon in its First set, insert all the Follow sets in the temp map with derivation " synch ".

Algorithm:

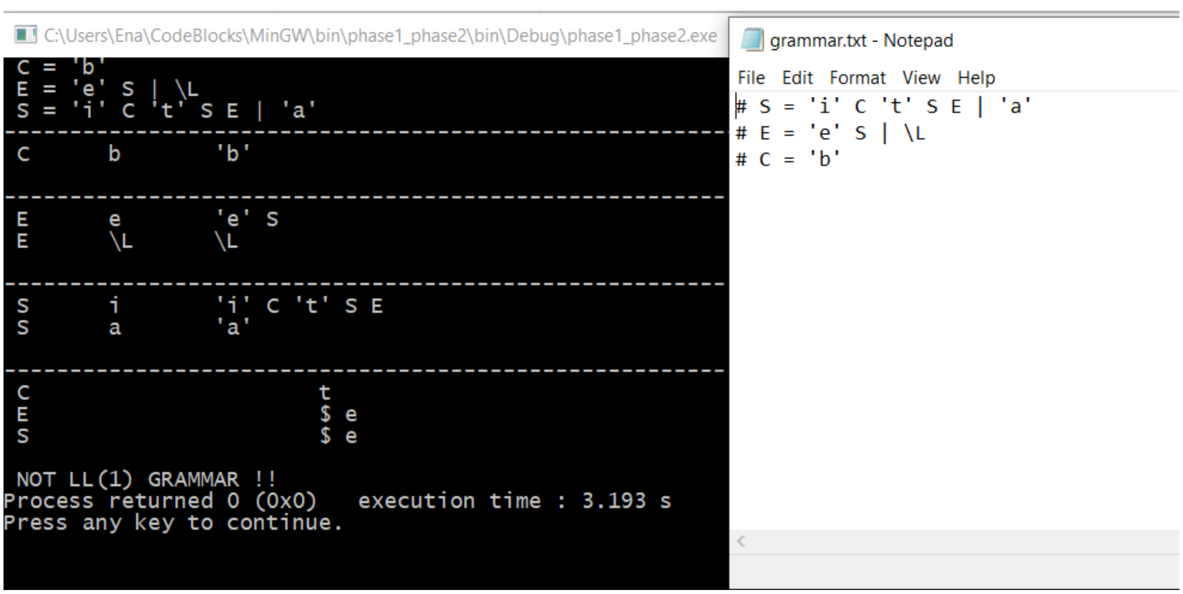
★ Call **get_Parsing_Table ()** using the First and Follow sets.

Assumptions:

★ The empty entry is **not stored** in the map.

Sample Runs:

★ This example **is not** LL1 grammar as under terminal e the non terminal E has 2 derivations.



★ **The provided grammar in the pdf :**

```
****************************
    Parsing Table is:
****************************

ASSIGNMENT
------------------------
       Terminal         Derivation
       $                synch
       float            synch
       id               'id' '=' EXPRESSION ';'
       if               synch
       int              synch
       while            synch
       }                synch
DECLARATION
------------------------
       Terminal         Derivation
       $                synch
       float            PRIMITIVE_TYPE 'id' ';'
       id               synch
       if               synch
       int              PRIMITIVE_TYPE 'id' ';'
       while            synch
       }                synch
EXPRESSION
------------------------
       Terminal         Derivation
       (                SIMPLE_EXPRESSION EXPRESSION1
       )                synch
       +                SIMPLE_EXPRESSION EXPRESSION1
       -                SIMPLE_EXPRESSION EXPRESSION1
       ;                synch
       id               SIMPLE_EXPRESSION EXPRESSION1
       num              SIMPLE_EXPRESSION EXPRESSION1
EXPRESSION1
------------------------
       Terminal         Derivation
       )                \L
       ;                \L
       relop            'relop' SIMPLE_EXPRESSION
```

```
FACTOR
-----------------------
        Terminal        Derivation
        (               '(' EXPRESSION ')'
        )               synch
        ;               synch
        addop           synch
        id              'id'
        mulop           synch
        num             'num'
        relop           synch
IF
-----------------------
        Terminal        Derivation
        $               synch
        float           synch
        id              synch
        if              'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
        int             synch
        while           synch
        }               synch
METHOD_BODY
-----------------------
        Terminal        Derivation
        $               synch
        float           STATEMENT_LIST
        id              STATEMENT_LIST
        if              STATEMENT_LIST
        int             STATEMENT_LIST
        while           STATEMENT_LIST
PRIMITIVE_TYPE
-----------------------
        Terminal        Derivation
        float           'float'
        id              synch
        int             'int'
SIGN
-----------------------
        Terminal        Derivation
        (               synch
        +               '+'
        -               '-'
        id              synch
        num             synch
```

```
SIMPLE_EXPRESSION
-----------------------
        Terminal          Derivation
        (                 TERM SIMPLE_EXPRESSION~
        )                 synch
        +                 SIGN TERM SIMPLE_EXPRESSION~
        -                 SIGN TERM SIMPLE_EXPRESSION~
        ;                 synch
        id                TERM SIMPLE_EXPRESSION~
        num               TERM SIMPLE_EXPRESSION~
        relop             synch
SIMPLE_EXPRESSION~
-----------------------
        Terminal          Derivation
        )                 \L
        ;                 \L
        addop             'addop' TERM SIMPLE_EXPRESSION~
        relop             \L
STATEMENT
-----------------------
        Terminal          Derivation
        $                 synch
        float             DECLARATION
        id                ASSIGNMENT
        if                IF
        int               DECLARATION
        while             WHILE
        }                 synch
STATEMENT_LIST
-----------------------
        Terminal          Derivation
        $                 synch
        float             STATEMENT STATEMENT_LIST~
        id                STATEMENT STATEMENT_LIST~
        if                STATEMENT STATEMENT_LIST~
        int               STATEMENT STATEMENT_LIST~
        while             STATEMENT STATEMENT_LIST~
STATEMENT_LIST~
-----------------------
        Terminal          Derivation
        $                 \L
        float             STATEMENT STATEMENT_LIST~
        id                STATEMENT STATEMENT_LIST~
        if                STATEMENT STATEMENT_LIST~
        int               STATEMENT STATEMENT_LIST~
        while             STATEMENT STATEMENT_LIST~
```

```
TERM
-----------------------
        Terminal          Derivation
        (                 FACTOR TERM~
        )                 synch
        ;                 synch
        addop             synch
        id                FACTOR TERM~
        num               FACTOR TERM~
        relop             synch
TERM~
-----------------------
        Terminal          Derivation
        )                 \L
        ;                 \L
        addop             \L
        mulop             'mulop' FACTOR TERM~
        relop             \L
WHILE
-----------------------
        Terminal          Derivation
        $                 synch
        float             synch
        id                synch
        if                synch
        int               synch
        while             'while' '(' EXPRESSION ')' '{' STATEMENT '}'
        }                 synch
```

Non Recursive Predictive Parsing -- LL(1) Parser

Data structures:

- **stack<string> Main_stack,** contains the grammar symbols, at the bottom of the stack, there is a special **end marker symbol $**.
- **map<string,map<string,string>> Parsing_Table,** which represent as :
  - ➔ Two-dimensional array .
  - ➔ Each row is a non-terminal symbol
  - ➔ Each column is a terminal symbol or the special symbol $.
  - ➔ Each entry holds a production rule.
- These data structure paths as a parameter to the **Parser_Handle** function.

Functions:

- **Parser_Handle (Parsing_Table, *Main_Stack, One_Token)** : check class
  - ➢ The main purpose of this function is to produce a production rule representing a step of the derivation sequence **(left-most derivation)** of the string in the input buffer.
  - ➢ The function takes a pointer on the **Main_stack**, **Parsing_Table and only one_token** at a time.
  - ➢ Set two flags as true :
    - one indicates if the sequence of input is accepted or not.
    - Another one to use at trigger in the main while loop in the function.
  - ➢ In the while loop (true) , store the Top of the stack in a temper variable then check if the **one_token** and **Top** equal “$” then check if sequence of input is accepted or not.
  - ➢ check if **Top** is terminal or not by searching in the rows of **Parsing_Table.**
  - ➢ If **Non Terminal** looks at the parsing table entry **Parsing_Table** [**Non_Terminal_variable**, **token**].
  - ➢ If **Parsing_Table** [**Non_Terminal_variable**, **token**] holds a production rule XY1Y2...Yk, it pops Non_Terminal_variable from the stack and pushes Yk,Yk-1,...,Y1 into the stack after splitting the production variables.
  - ➢ The parser also outputs the production rule XY1Y2...Yk to **represent a step of the derivation**.

- ➢ If **Parsing_Table** [**Non_Terminal_variable**, **token**] holds "synch", representing All the terminal-symbols in the following set of a non-terminal the parser pops the Non_Terminal_variable and outputs the **Ignore error**.
- ➢ If the **entry is empty,** output an error and discard the token by breaking the while loop by setting the flag false and take another one by calling the **get_next_token()** function.
- ➢ if **Terminal** compares with the **One_token :**
  - If an equal parser pops Terminal_variable from the stack, and moves the next symbol in the input buffer.
  - If a different Report Error and if Terminal_variable is not equal "$" set the sequence of token is not accepted and parser pops Terminal_variable from the stack.

❑ **get_next_token() :** main class

- ➢ The main function in the parser phase calls the lexical phase to get the next token.
- ➢ In this function take one word from the file then validate it by using the final graph to represent the minimized DFA graph.
- ➢ Check if the token equals "=" parse as "assign".
- ➢ If the words in the file finished parse the "$".
- ➢ return a new token as string .

Algorithm:

★ Call **generate_final_output**(start_grammar, Parsing_Table) which creates the main stack and pushes on it the "$" and start_grammar and then calls the Parser_Handle function.

Assumptions:

★ The only case is that the sequence of the token is not accepted when the terminal variable is not an equal token.

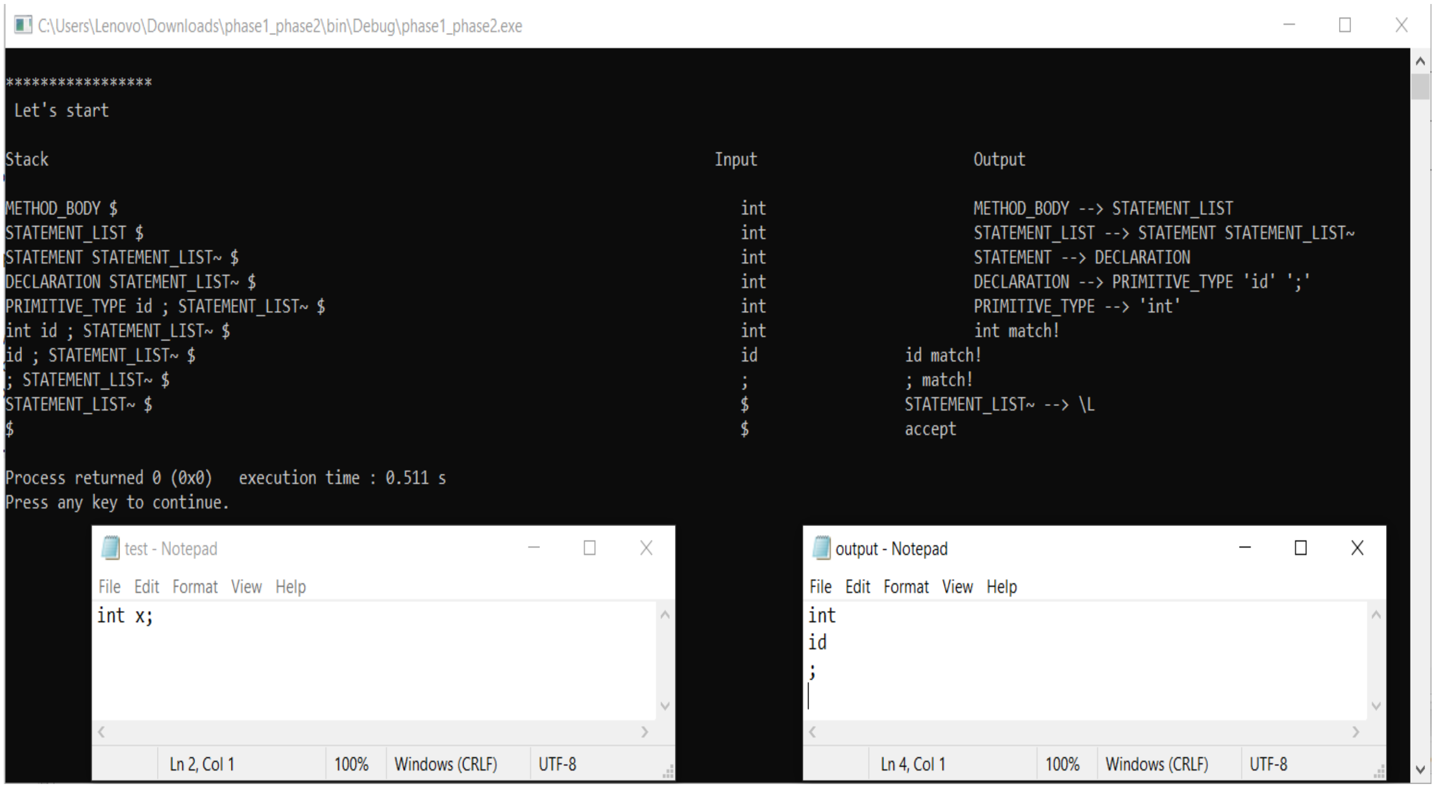
★ Transition table in Minimal DFA :

```
DFA After minimization
*******************

state id is: 1
Its accepting state is 0
under input !  it goes to state 2
under input *  it goes to state 3
under input +  it goes to state 4
under input -  it goes to state 4
under input /  it goes to state 3
under input 0-9  it goes to state 7
under input <  it goes to state 8
under input =  it goes to state 9
under input >  it goes to state 8
under input A-Z  it goes to state 11
under input a-z  it goes to state 11
*******************

state id is: 2
Its accepting state is 0
under input =  it goes to state 13
*******************

state id is: 3
Its accepting state is 1
*******************

state id is: 4
Its accepting state is 1
*******************

state id is: 7
Its accepting state is 1
under input .  it goes to state 14
under input 0-9  it goes to state 7
*******************

state id is: 8
Its accepting state is 1
under input =  it goes to state 13
*******************

state id is: 9
Its accepting state is 1
under input =  it goes to state 13
*******************
```

```
state id is: 11
Its accepting state is 1
under input 0-9  it goes to state 11
under input A-Z  it goes to state 11
under input a-z  it goes to state 11
*******************

state id is: 13
Its accepting state is 1
*******************

state id is: 14
Its accepting state is 0
under input 0-9  it goes to state 21
*******************

state id is: 21
Its accepting state is 1
under input 0-9  it goes to state 21
under input E  it goes to state 22
*******************

state id is: 22
Its accepting state is 0
under input 0-9  it goes to state 23
*******************

state id is: 23
Its accepting state is 1
under input 0-9  it goes to state 23
*******************
```

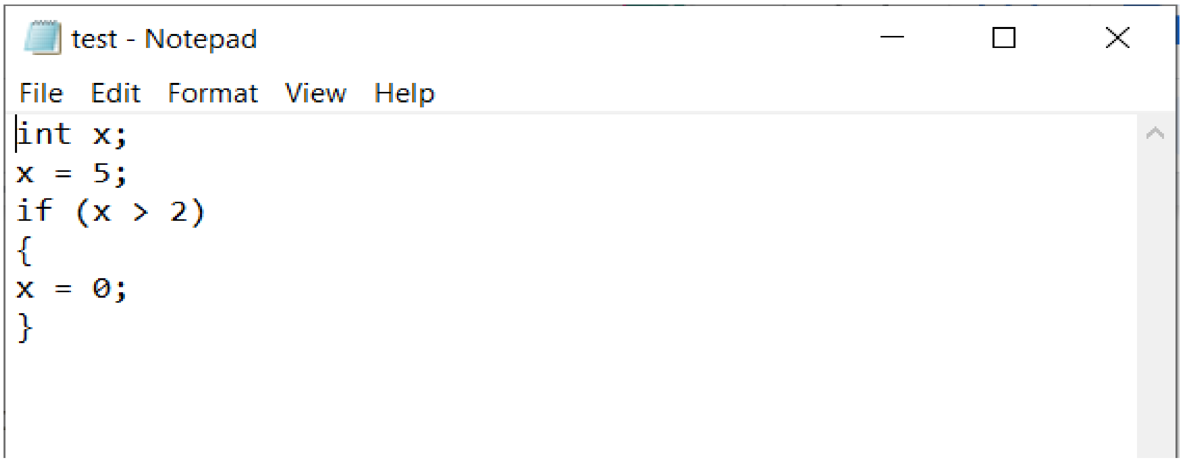
Sample Runs:

★ Example 1 : simple example



★ Example 2 : example provided in phase 2 pdf :

- The test program :

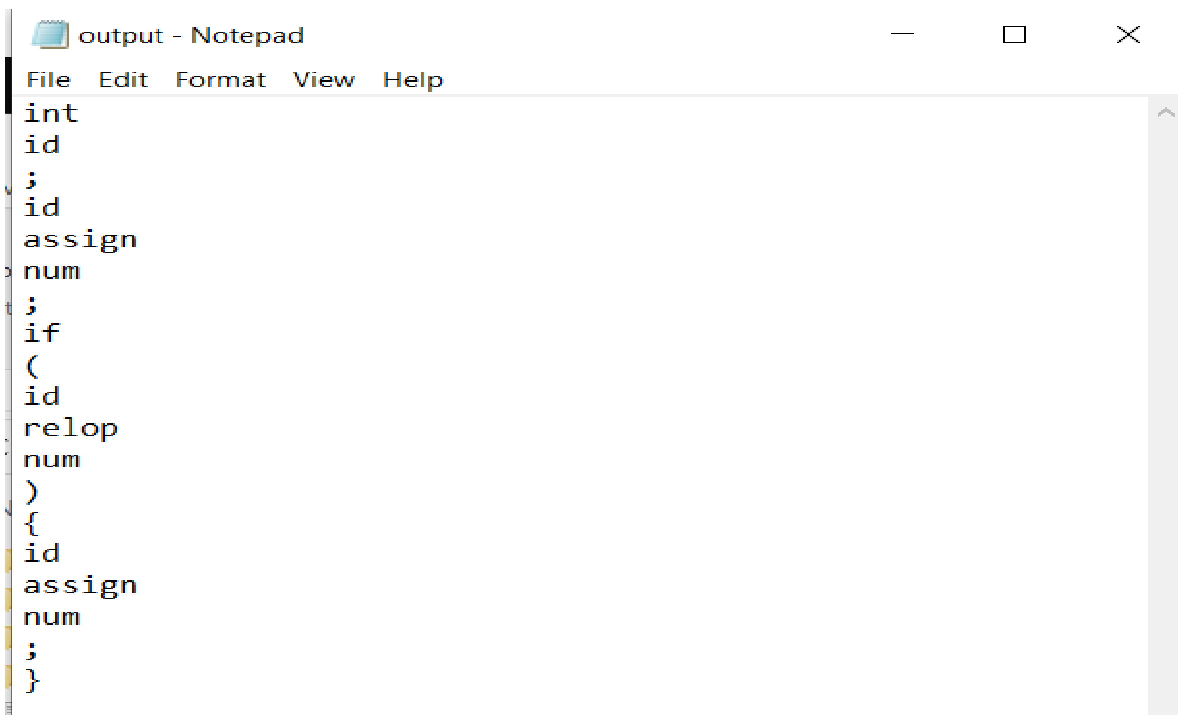


test - Notepad

File Edit Format View Help

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

- The output tokens from phase1 :



output - Notepad

File Edit Format View Help

```
int
id
;
id
assign
num
;
if
(
id
relop
num
)
{
id
assign
num
;
}
```

- The values in the stack :

```
METHOD_BODY $
STATEMENT_LIST $
STATEMENT STATEMENT_LIST~ $
DECLARATION STATEMENT_LIST~ $
PRIMITIVE_TYPE id ; STATEMENT_LIST~ $
int id ; STATEMENT_LIST~ $
id ; STATEMENT_LIST~ $
; STATEMENT_LIST~ $
STATEMENT_LIST~ $
STATEMENT STATEMENT_LIST~ $
ASSIGNMENT STATEMENT_LIST~ $
id = EXPRESSION ; STATEMENT_LIST~ $
= EXPRESSION ; STATEMENT_LIST~ $
EXPRESSION ; STATEMENT_LIST~ $
SIMPLE_EXPRESSION EXPRESSION1 ; STATEMENT_LIST~ $
TERM SIMPLE_EXPRESSION~ EXPRESSION1 ; STATEMENT_LIST~ $
FACTOR TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ; STATEMENT_LIST~ $
num TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ; STATEMENT_LIST~ $
TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ; STATEMENT_LIST~ $
SIMPLE_EXPRESSION~ EXPRESSION1 ; STATEMENT_LIST~ $
EXPRESSION1 ; STATEMENT_LIST~ $
; STATEMENT_LIST~ $
STATEMENT_LIST~ $
STATEMENT STATEMENT_LIST~ $
IF STATEMENT_LIST~ $
if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
SIMPLE_EXPRESSION EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
TERM SIMPLE_EXPRESSION~ EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
FACTOR TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
id TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
SIMPLE_EXPRESSION~ EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
relop SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
TERM SIMPLE_EXPRESSION~ ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
FACTOR TERM~ SIMPLE_EXPRESSION~ ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
num TERM~ SIMPLE_EXPRESSION~ ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
TERM~ SIMPLE_EXPRESSION~ ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
SIMPLE_EXPRESSION~ ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
) { STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
{ STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
STATEMENT } else { STATEMENT } STATEMENT_LIST~ $
ASSIGNMENT } else { STATEMENT } STATEMENT_LIST~ $
```

```
id = EXPRESSION ; } else { STATEMENT } STATEMENT_LIST~ $
= EXPRESSION ; } else { STATEMENT } STATEMENT_LIST~ $
EXPRESSION ; } else { STATEMENT } STATEMENT_LIST~ $
SIMPLE_EXPRESSION EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
TERM SIMPLE_EXPRESSION~ EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
FACTOR TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
num TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
TERM~ SIMPLE_EXPRESSION~ EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
SIMPLE_EXPRESSION~ EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST~ $
; } else { STATEMENT } STATEMENT_LIST~ $
} else { STATEMENT } STATEMENT_LIST~ $
else { STATEMENT } STATEMENT_LIST~ $
{ STATEMENT } STATEMENT_LIST~ $
STATEMENT } STATEMENT_LIST~ $
} STATEMENT_LIST~ $
STATEMENT_LIST~ $
$
```

- The output of phase 2 :

C:\Users\Ena\CodeBlocks\MinGW\bin\phase1_phase2\bin\Debug\phase1_phase2.exe

```
************************ Output ******************************
 METHOD_BODY --> STATEMENT_LIST
 STATEMENT_LIST --> STATEMENT STATEMENT_LIST~
 STATEMENT --> DECLARATION
 DECLARATION --> PRIMITIVE_TYPE 'id' ';'
 PRIMITIVE_TYPE --> 'int'
 int match!
 id match!
 ; match!
 STATEMENT_LIST~ --> STATEMENT STATEMENT_LIST~
 STATEMENT --> ASSIGNMENT
 ASSIGNMENT --> 'id' '=' EXPRESSION ';'
 id match!
 = match!
 EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION1
 SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
 TERM --> FACTOR TERM~
 FACTOR --> 'num'
 num match!
 TERM~ --> \L
 SIMPLE_EXPRESSION~ --> \L
 EXPRESSION1 --> \L
 ; match!
 STATEMENT_LIST~ --> STATEMENT STATEMENT_LIST~
 STATEMENT --> IF
 IF --> 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
 if match!
 ( match!
 EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION1
 SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
 TERM --> FACTOR TERM~
 FACTOR --> 'id'
 id match!
 TERM~ --> \L
 SIMPLE_EXPRESSION~ --> \L
 EXPRESSION1 --> 'relop' SIMPLE_EXPRESSION
 relop match!
 SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
 TERM --> FACTOR TERM~
 FACTOR --> 'num'
 num match!
 TERM~ --> \L
 SIMPLE_EXPRESSION~ --> \L
 ) match!
 { match!
 STATEMENT --> ASSIGNMENT
 ASSIGNMENT --> 'id' '=' EXPRESSION ';'
```

```
id match!
= match!
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION1
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> 'num'
num match!
TERM~ --> \L
SIMPLE_EXPRESSION~ --> \L
EXPRESSION1 --> \L
; match!
} match!
Error: missing else, inserted
Error: missing {, inserted
Ignore Error:(illegal STATEMENT)
Error: missing }, inserted
STATEMENT_LIST~ --> \L
accept

Process returned 0 (0x0)   execution time : 2.432 s
Press any key to continue.
```

★ Example 3 : example provided in phase 1 pdf :

- The test program and the output of phase 1 :

output.txt - Notepad

```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

test.txt - Notepad

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

Ln 1, Col 1 | 100% | Windows (CRLF

- ## The output of phase 2 :

C:\Users\Ena\CodeBlocks\MinGW\bin\phase1_phase2\bin\Debug\phase1_phase2.exe

```
************************ Output of parser phase****************************

METHOD_BODY --> STATEMENT_LIST
STATEMENT_LIST --> STATEMENT STATEMENT_LIST~
STATEMENT --> DECLARATION
DECLARATION --> PRIMITIVE_TYPE 'id' ';'
PRIMITIVE_TYPE --> 'int'
int match!
id match!
Error: missing ;, inserted
Error:(illegal STATEMENT_LIST~) - discard ,
STATEMENT_LIST~ --> STATEMENT STATEMENT_LIST~
STATEMENT --> ASSIGNMENT
ASSIGNMENT --> 'id' '=' EXPRESSION ';'
id match!
Error: missing =, inserted
Error:(illegal EXPRESSION) - discard ,
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION1
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> 'id'
id match!
Error:(illegal TERM~) - discard ,
Error:(illegal TERM~) - discard id
TERM~ --> \L
SIMPLE_EXPRESSION~ --> \L
EXPRESSION1 --> \L
; match!
STATEMENT_LIST~ --> STATEMENT STATEMENT_LIST~
STATEMENT --> WHILE
WHILE --> 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
while match!
( match!
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION1
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> 'id'
id match!
TERM~ --> \L
SIMPLE_EXPRESSION~ --> \L
EXPRESSION1 --> 'relop' SIMPLE_EXPRESSION
relop match!
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> 'num'
num match!
TERM~ --> \L
```

```
TERM~ --> \L
SIMPLE_EXPRESSION~ --> \L
) match!
{ match!
STATEMENT --> ASSIGNMENT
ASSIGNMENT --> 'id' '=' EXPRESSION ';'
id match!
= match!
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION1
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> 'id'
id match!
TERM~ --> \L
SIMPLE_EXPRESSION~ --> 'addop' TERM SIMPLE_EXPRESSION~
addop match!
TERM --> FACTOR TERM~
FACTOR --> 'num'
num match!
TERM~ --> \L
SIMPLE_EXPRESSION~ --> \L
EXPRESSION1 --> \L
; match!
} match!
STATEMENT_LIST~ --> \L
accept

Process returned 0 (0x0)   execution time : 0.952 s
Press any key to continue.
```