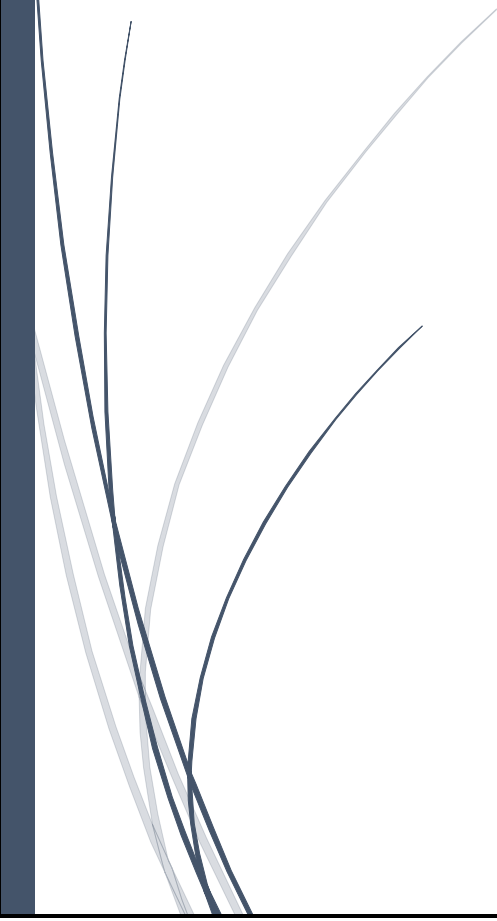




## Compiler Phase 2

### Names:

- Ereny Zarif Abd El Shaheed ID: 18
  - Maria Onsy Salib ID: 51
  - Nada Fathy Ali ID: 67
- 

# Parser

## Data structure:

- **Node:** having
  - string name: the name of the node.
  - bool terminal: if true then it's terminal, otherwise it's nonterminal.
  - Node represents the right side of each rule.
- **Rule:** having
  - int id: id associated to this rule.
  - int non\_terminal: id of the nonterminal of each rule (head).
  - list of nodes: each node contains one of terminals or nonterminal.

One production rule can have more than one rule (if it has more than one production "or").

- **Non\_terminal:** having
  - string name: name of nonterminal.
  - int id: id associated to this nonterminal.
  - list of rules: containing all the productions for this nonterminal.
  - list of (pair of int) first: this is the first set of nonterminal, each pair contains id of terminal and id of production.
  - List of int follow: this is the follow set of nonterminal, contains id of terminals.
- **Grammar:** having
  - string filename: path of file that contains the grammar productions.
  - list of Non\_terminal non\_terminals: containing all nonterminal given in the grammar.
  - list of string terminals: containing all terminals given in the grammar.
  - list of rule allRules: containing all the rules in the given grammar.
  - Int numNonterm: number of nonterminals.
  - Int numRule: number of rules.

- **Parser\_table:** having
  - Grammer
  - List of (list of int) table: this is the parse table.
  - Bool nonLL1: to indicate that the given grammer is LL1 or not.
  - String error\_message: the error message if the grammar is not LL1.
- **Parser:** having
  - Grammar\* g
  - Link linker = Link ()
  - list of (list of int) table: this is the parse table.
  - List of string output: the output to be printed in the output file.
  - Stack of pair of (int, bool) st: the int determine the id of the terminal or the nonterminal, the bool indicate if it is a terminal or not.
  - int input: the id of the terminal which is the current input.
  - bool getNext: indicate if we have to get the next token or not.
- **Linker:** having
  - Rules rules
  - Final\_NDFA ndfa
  - DFA dfa
  - Minimization mini
  - Tokens tk

To link lexical generator with parser.


## Algorithms:

- To compute first set:
  - Iterate over the rules of this nonterminal
    - 1) If the first node in the rule is a terminal add it to the first list of this nonterminal with bool equals true.
    - 2) If the first node in the rule is a nonterminal
      - If the first list of the nonterminal is not empty add it to the first list of that nonterminal.
      - If the first list of the nonterminal is empty compute it then add it to the first list of that nonterminal.
- To compute follow set:
  - Check if the nonterminal is start symbol, then add \$ sign to the follow set.
  - Iterate over all rules:
    - 1) check if that rule contains nonterminal (say e) and not the left most nonterminal
      - check if the next one is terminal, add its id to the follow set of e.
      - If nonterminal (say s), add its first set of s to the follow set of e and checks if the first set of s contain epsilon, then add follow set of the nonterminal (head) of the rule to the follow set of e.
    - 2) check if the rule contains e and it's the left most nonterminal
      - add the follow set of nonterminal (head) of the rule to the follow set of e.
- To construct parse table:
  - Iterate over all nonterminals, for every nonterminal:
    - Iterate over first set, for each terminal in the first set add its production id to the table in entry of (row: nonterminal id, column: terminal id).
    - If the first set contains epsilon, then iterate over follow set of that nonterminal and add epsilon production to the table in entry of (row: nonterminal id, column: terminal id).
    - Else iterate over follow set and add id of synch to indicate that this is a synchronization token.

- To drive the left most derivation for an input:
  - Push -1 (which indicate "\$") to the stack then push the start nonterminal id with bool false to indicate that this is a nonterminal.
  - If the top of the stack is a terminal:
    - Check if it equals the input pop it from stack and let the bool getNext is true.
    - Else indicate error to add the missing terminal to the input and pop it from the stack
  - If the top of the stack is nonterminal:
    - Get the int in the parser table at index of the id of nonterminal which is at the top of the stack and the id of the terminal which is the input.
    - If the int is -3 (indicate error) discard the current input and let the bool getNext is true.
    - If the int is -2 (indicate synchronize) pop this nonterminal from the stack and parse the next top of the stack.
    - Else get the rule which its id equals to that int, pop the top of the stack and then push the nodes of the rule in reverse order. Parse again.
    - If the stack is empty and also the input is empty return.
    - Else if the stack is empty and also the input is not or if the input is empty and the stack is not, indicate error.
- To get the next token:
  - Class link is used to link between the lexical generator and the parser.
  - By calling the function get\_next\_token in link it will call the get\_next\_token function in the lexical which return the next token.

# Output

- Parser table:

 Parser Table.txt - Notepad

File Edit Format View Help

```
Nonterminal:    METHOD_BODY
\L -->  error
id -->  METHOD_BODY = STATEMENT_LIST
; -->  error
int -->  METHOD_BODY = STATEMENT_LIST
float -->  METHOD_BODY = STATEMENT_LIST
if -->  METHOD_BODY = STATEMENT_LIST
( -->  error
) -->  error
{ -->  error
} -->  error
else -->  error
while -->  METHOD_BODY = STATEMENT_LIST
= -->  error
relop -->  error
addop -->  error
mulop -->  error
num -->  error
+ -->  error
- -->  error
$ -->  sync
-----
Nonterminal:    STATEMENT_LIST
\L -->  error
id -->  STATEMENT_LIST = STATEMENT STATEMENT_LIST1
; -->  error
int -->  STATEMENT_LIST = STATEMENT STATEMENT_LIST1
float -->  STATEMENT_LIST = STATEMENT STATEMENT_LIST1
if -->  STATEMENT_LIST = STATEMENT STATEMENT_LIST1
( -->  error
) -->  error
{ -->  error
} -->  error
else -->  error
```

```
else --> error
while --> STATEMENT_LIST = STATEMENT STATEMENT_LIST1
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> sync
```

-----

Nonterminal:       STATEMENT\_LIST1

```
\L --> error
id --> STATEMENT_LIST1 = STATEMENT STATEMENT_LIST1
; --> error
int --> STATEMENT_LIST1 = STATEMENT STATEMENT_LIST1
float --> STATEMENT_LIST1 = STATEMENT STATEMENT_LIST1
if --> STATEMENT_LIST1 = STATEMENT STATEMENT_LIST1
( --> error
) --> error
{ --> error
} --> error
else --> error
while --> STATEMENT_LIST1 = STATEMENT STATEMENT_LIST1
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> STATEMENT_LIST1 = '\L'
```

-----

Nonterminal:       STATEMENT

\\

```

-----
Nonterminal:      STATEMENT
\L --> error
id --> STATEMENT = ASSIGNMENT
; --> error
int --> STATEMENT = DECLARATION
float --> STATEMENT = DECLARATION
if --> STATEMENT = IF
( --> error
) --> error
{ --> error
} --> sync
else --> error
while --> STATEMENT = WHILE
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> sync
-----

```

```

-----
Nonterminal:      DECLARATION
\L --> error
id --> sync
; --> error
int --> DECLARATION = PRIMITIVE_TYPE 'id' ';'
float --> DECLARATION = PRIMITIVE_TYPE 'id' ';'
if --> sync
( --> error
) --> error
{ --> error
} --> sync
else --> error
while --> sync
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> sync
-----

```



```

-----
Nonterminal:    PRIMITIVE_TYPE
\L --> error
id --> sync
; --> error
int --> PRIMITIVE_TYPE = 'int'
float --> PRIMITIVE_TYPE = 'float'
if --> error
( --> error
) --> error
{ --> error
} --> error
else --> error
while --> error
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> error
-----

```

```

-----
Nonterminal:    IF
\L --> error
id --> sync
; --> error
int --> sync
float --> sync
if --> IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
( --> error
) --> error
{ --> error
} --> sync
else --> error
while --> sync
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> sync
-----

```

```

-----
Nonterminal:      WHILE
\L --> error
id --> sync
; --> error
int --> sync
float --> sync
if --> sync
( --> error
) --> error
{ --> error
} --> sync
else --> error
while --> WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> sync
-----

```

```

-----
Nonterminal:      ASSIGNMENT
\L --> error
id --> ASSIGNMENT = 'id' '=' EXPRESSION ';'
; --> error
int --> sync
float --> sync
if --> sync
( --> error
) --> error
{ --> error
} --> sync
else --> error
while --> sync
= --> error
relop --> error
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> sync
-----

```

```

-----
Nonterminal:      EXPRESSION
\L --> error
id --> EXPRESSION = SIMPLE_EXPRESSION EXPRESSION1
; --> sync
int --> error
float --> error
if --> error
( --> EXPRESSION = SIMPLE_EXPRESSION EXPRESSION1
) --> sync
{ --> error
} --> error
else --> error
while --> error
= --> error
relop --> error
addop --> error
mulop --> error
num --> EXPRESSION = SIMPLE_EXPRESSION EXPRESSION1
+ --> EXPRESSION = SIMPLE_EXPRESSION EXPRESSION1
- --> EXPRESSION = SIMPLE_EXPRESSION EXPRESSION1
$ --> error
-----

```

```

-----
Nonterminal:      EXPRESSION1
\L --> error
id --> error
; --> EXPRESSION1 = '\L'
int --> error
float --> error
if --> error
( --> error
) --> EXPRESSION1 = '\L'
{ --> error
} --> error
else --> error
while --> error
= --> error
relop --> EXPRESSION1 = 'relop' SIMPLE_EXPRESSION
addop --> error
mulop --> error
num --> error
+ --> error
- --> error
$ --> error
-----

```

```

-----
Nonterminal:      SIMPLE_EXPRESSION
\L -->  error
id -->  SIMPLE_EXPRESSION = TERM SIMPLE_EXPRESSION1
; -->  sync
int -->  error
float -->  error
if -->  error
( -->  SIMPLE_EXPRESSION = TERM SIMPLE_EXPRESSION1
) -->  sync
{ -->  error
} -->  error
else -->  error
while -->  error
= -->  error
relop -->  sync
addop -->  error
mulop -->  error
num -->  SIMPLE_EXPRESSION = TERM SIMPLE_EXPRESSION1
+ -->  SIMPLE_EXPRESSION = SIGN TERM SIMPLE_EXPRESSION1
- -->  SIMPLE_EXPRESSION = SIGN TERM SIMPLE_EXPRESSION1
$ -->  error
-----

```

```

-----
Nonterminal:      SIMPLE_EXPRESSION1
\L -->  error
id -->  error
; -->  SIMPLE_EXPRESSION1 = '\L'
int -->  error
float -->  error
if -->  error
( -->  error
) -->  SIMPLE_EXPRESSION1 = '\L'
{ -->  error
} -->  error
else -->  error
while -->  error
= -->  error
relop -->  SIMPLE_EXPRESSION1 = '\L'
addop -->  SIMPLE_EXPRESSION1 = 'addop' TERM SIMPLE_EXPRESSION1
mulop -->  error
num -->  error
+ -->  error
- -->  error
$ -->  error
-----

```

-----  
Nonterminal:     TERM  
\L --> error  
id --> TERM = FACTOR TERM1  
; --> sync  
int --> error  
float --> error  
if --> error  
( --> TERM = FACTOR TERM1  
) --> sync  
{ --> error  
} --> error  
else --> error  
while --> error  
= --> error  
relop --> sync  
addop --> sync  
mulop --> error  
num --> TERM = FACTOR TERM1  
+ --> error  
- --> error  
\$ --> error  
-----

-----  
Nonterminal:     TERM1  
\L --> error  
id --> error  
; --> TERM1 = '\L'  
int --> error  
float --> error  
if --> error  
( --> error  
) --> TERM1 = '\L'  
{ --> error  
} --> error  
else --> error  
while --> error  
= --> error  
relop --> TERM1 = '\L'  
addop --> TERM1 = '\L'  
mulop --> TERM1 = 'mulop' FACTOR TERM1  
num --> error  
+ --> error  
- --> error  
\$ --> error  
-----

-----  
Nonterminal:      FACTOR

\L --> error

id --> FACTOR = 'id'

; --> sync

int --> error

float --> error

if --> error

( --> FACTOR = '(' EXPRESSION ')'

) --> sync

{ --> error

} --> error

else --> error

while --> error

= --> error

relop --> sync

addop --> sync

mulop --> sync

num --> FACTOR = 'num'

+ --> error

- --> error

\$ --> error  
-----

-----  
Nonterminal:      SIGN

\L --> error

id --> sync

; --> error

int --> error

float --> error

if --> error

( --> sync

) --> error

{ --> error

} --> error

else --> error

while --> error

= --> error

relop --> error

addop --> error

mulop --> error


num --> sync

+ --> SIGN = '+'

- --> SIGN = '-'

\$ --> error  
-----

## • Parser Output:

 Parser Output.txt - Notepad

File Edit Format View Help

```
METHOD_BODY
STATEMENT_LIST
STATEMENT STATEMENT_LIST1
DECLARATION STATEMENT_LIST1
PRIMITIVE_TYPE id ; STATEMENT_LIST1
int id ; STATEMENT_LIST1
id ; STATEMENT_LIST1
; STATEMENT_LIST1
STATEMENT_LIST1
STATEMENT STATEMENT_LIST1
ASSIGNMENT STATEMENT_LIST1
id = EXPRESSION ; STATEMENT_LIST1
= EXPRESSION ; STATEMENT_LIST1
EXPRESSION ; STATEMENT_LIST1
SIMPLE_EXPRESSION EXPRESSION1 ; STATEMENT_LIST1
TERM SIMPLE_EXPRESSION1 EXPRESSION1 ; STATEMENT_LIST1
FACTOR TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; STATEMENT_LIST1
num TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; STATEMENT_LIST1
TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; STATEMENT_LIST1
\L SIMPLE_EXPRESSION1 EXPRESSION1 ; STATEMENT_LIST1
SIMPLE_EXPRESSION1 EXPRESSION1 ; STATEMENT_LIST1
\L EXPRESSION1 ; STATEMENT_LIST1
EXPRESSION1 ; STATEMENT_LIST1
\L ; STATEMENT_LIST1
; STATEMENT_LIST1
STATEMENT_LIST1
STATEMENT STATEMENT_LIST1
IF STATEMENT_LIST1
if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
SIMPLE_EXPRESSION EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
TERM SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
FACTOR TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
```

```
FACTOR TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
id TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
\L SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
SIMPLE_EXPRESSION1 EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
\L EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
relop SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
TERM SIMPLE_EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
FACTOR TERM1 SIMPLE_EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
num TERM1 SIMPLE_EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
TERM1 SIMPLE_EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
\L SIMPLE_EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
SIMPLE_EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
\L ) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
) { STATEMENT } else { STATEMENT } STATEMENT_LIST1
{ STATEMENT } else { STATEMENT } STATEMENT_LIST1
STATEMENT } else { STATEMENT } STATEMENT_LIST1
ASSIGNMENT } else { STATEMENT } STATEMENT_LIST1
id = EXPRESSION ; } else { STATEMENT } STATEMENT_LIST1
= EXPRESSION ; } else { STATEMENT } STATEMENT_LIST1
EXPRESSION ; } else { STATEMENT } STATEMENT_LIST1
SIMPLE_EXPRESSION EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
TERM SIMPLE_EXPRESSION1 EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
FACTOR TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
num TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
\L SIMPLE_EXPRESSION1 EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
SIMPLE_EXPRESSION1 EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
\L EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST1
\L ; } else { STATEMENT } STATEMENT_LIST1
; } else { STATEMENT } STATEMENT_LIST1
```



```

\l ; } else { STATEMENT } STATEMENT_LIST1
; } else { STATEMENT } STATEMENT_LIST1
} else { STATEMENT } STATEMENT_LIST1
{ STATEMENT } STATEMENT_LIST1
STATEMENT } STATEMENT_LIST1
ASSIGNMENT } STATEMENT_LIST1
id = EXPRESSION ; } STATEMENT_LIST1
= EXPRESSION ; } STATEMENT_LIST1
EXPRESSION ; } STATEMENT_LIST1
SIMPLE_EXPRESSION EXPRESSION1 ; } STATEMENT_LIST1
TERM SIMPLE_EXPRESSION1 EXPRESSION1 ; } STATEMENT_LIST1
FACTOR TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; } STATEMENT_LIST1
num TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; } STATEMENT_LIST1
TERM1 SIMPLE_EXPRESSION1 EXPRESSION1 ; } STATEMENT_LIST1
\l SIMPLE_EXPRESSION1 EXPRESSION1 ; } STATEMENT_LIST1
SIMPLE_EXPRESSION1 EXPRESSION1 ; } STATEMENT_LIST1
\l EXPRESSION1 ; } STATEMENT_LIST1
EXPRESSION1 ; } STATEMENT_LIST1
\l ; } STATEMENT_LIST1
; } STATEMENT_LIST1
} STATEMENT_LIST1
STATEMENT_LIST1
\l

```