Compiler

Phase II

Names:

1 – Aya Osama Foad

26 – Salma Abd El-Aziz

27 – Salma Mohamed

29 – Shadwa Abd El-Mobdy

Prefix

1. Data structures

1.1 Basic data structures

- *Non-terminal*

Map <string, vector<vector<string>>> to keep all non-terminals associated with their productions.

- *Terminal*

Map <string, string> to keep all terminal to determine if this is terminal or not in O (1).

- *Nonter*

Vector <string> to keep all non-terminals in order to iterate on them.

1.2 Data structures associated with first algorithm

- *Map*
  - Map with a string as a key (non-terminal) and a vector of string as a value to preserve the first sets of each Nonterminal.
  - Map with a string as a key (non-terminal) and a vector of string as a value to preserve the first sets of each non-terminal.

1.3 Data structures associated with follow algorithm

- *Struct 'followGp'*

Consists of four elements which helps us to construct the initial table for each non-terminal:

  - A vector to hold non-terminals which we will include its first terminals in the current non-terminal follow.

- A vector to hold non-terminals which we will include its followers in the current non-terminal follow.
- A vector to hold terminals which we will in the current non-terminal follow.
- A Boolean to decide whether the follow of the current non-terminal has completely calculated or not.

- *Map*
  - <string, followGp> productionFollowers : it is a map that holds the initial table, its key is the non-terminal and its value is the 'followGp' struct which we have previously discussed.
  - <string, vector<string>> follow : it is a map that holds the final table, its key is the non-terminal and its value is a vector of terminal followers for that non-terminal.

1.4 Data structures associated with parsing algorithm

- *Track*

Stack <string> to keep track the left most derivation.

2. Algorithm

2.1 Left factoring

```
left_factoring () {
    for each non-terminal
        call handle_productions method.
}

handle_productions (vector<string>::iterator n_ter) {
    call take_commons method for this non-terminal
    call non_immediate_LF method for this non-terminal
}

take_commons (vector<string>::iterator n_ter) {
    if this non-terminal has one production
        no need for check so return.
    calculate first for this non-terminal to decide commons in which
    productions.
    for each common factor
        if it participates into only one production
            continue.
        add new non-terminal to basic data structures.
        create its productions.
        remove unneeded productions from the original non-terminal.
        update original non-terminal with the new production containing the new
        created non-terminal.
        recurse on the new non-terminal to take all nested common factors.
}

non_immediate_LF (string n_ter, bool flag) {
    check if no more than one production or flag is asserted
        return.
    calculate firsts of all productions even if productions starting with
    non-terminals to decide if the will make common factors with other
    productions.
    substitute in productions strting with non-terminals if they whould make
    common factors then call take_commons method to handle the left factoring
    that resulted from substituting.
    recurse on the same non-terminal if substituing was not the final
    substituting to handle nested productions with flag is asserted or not
    depends on what happened in substituting.
}
```

2.2 Left recursion

```
leftRecursion()
    begin
        for i from 1 to nonterminal.size() do
            for 1 to i-1 do
                replace each production
                Ai = Aj y
                by
                Ai = x1y|...|xmy
                where
                Aj = x1|...|xm
            eliminate immediate left-recursions among Ai
    end
```

2.3 First

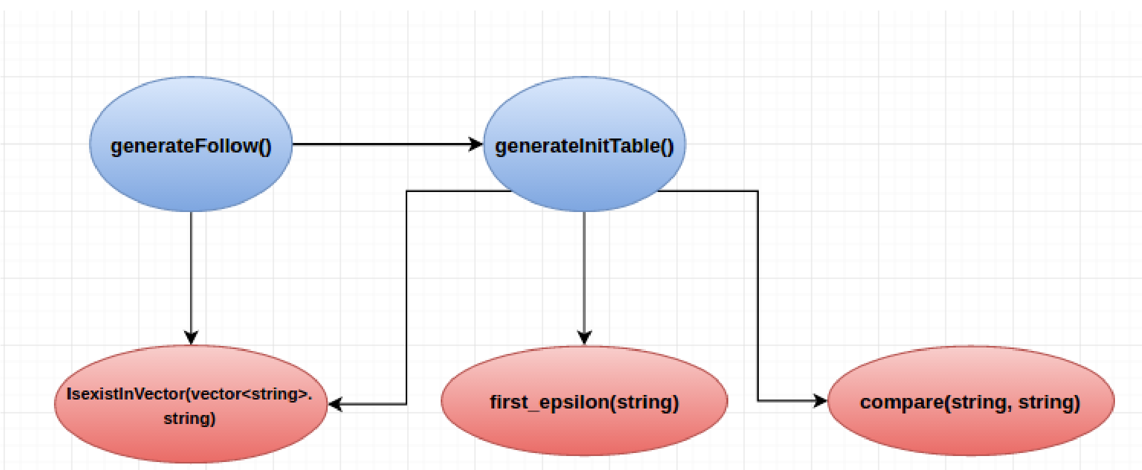
```
readGrammar()
    begin
        open input file
        for each line
            if(match pattern for new production)
                add new Nonterminal associated with its new productions
            else if(match pattern for extra producions)
                get previous Nonterminal's productions
                push extra productions into it
            else
                print"Error Line!!"
        end for

        close file
        call leftFactoring()
        call eliminateLeftRecursion()
    end
```

```
calc_firsts() {
    finding first for each nonterminal, by calling calc_first() and giving it
    the nonterminal calc_first().
}

calc_first(nonterm, prev_nonterm) {
    // input : nonterm is the nonterminal which we calculate its first,
       prev_nonterm is the nonterminal that called the method.
    // thit function fills a map that holds the first sets of each nonterminal.
    for each production rule in nonterm do
        if the production rule starts with a terminal
            add it to the first set.
        else if the production rule starts with a EPSILON
            add it to the first set.
        else the production rule starts with a nonterminal N,
            repeate.
            check if N's first set is calculated (by checking the map), if it
            wasn't calculated
                call calc_first(N, nontrem).
                add N's first set in nonterm's first.
                take the next token in the production rule.
            while (N's first set contains EPSILON & production rule doesn't
            finish)
            if the last token contains EPSILON
                add EPSILON in nonterm's first set.
    add nonterm's first set in the map.
}
```

2.4 Follow



```
int IsexistInVector(vector<string>, string):
--------------------------------------------
This method takes a vector of strings and a string and it checks whether this
string exists in this vector or not, if the string exists it will return its
index in the vector, else it will return -1.

int compare(string, string):
----------------------------
This method takes two strings (Non-Terminals) and compare whether the first
string comes after the second string or not, if so it will return 1, else
it will return 0.

bool first_epsilon(string):
---------------------------
This method takes a string (Non-Terminal) and return true if its first has
epsilon, if its first doesnot contain epsilon it will return false;
```

```
void generateInitTable():
-------------------------
This method is responsible for generating the initial table by filling the
map<string, followGp>productionFollowers, using the
map<string,vector<vector<string>>> nonterminal which contains the grammar
rules, the Non-Terminal as a key and the productions as vector of vector.
First: we iterate on each grammar rule,
Second: we iterate on each production in the the grammar rule,
Third: we loop on each element (Terminal/Non-Terminal) in the production
    we have two paths:
    First (if this element is Termainal): Skip it.
    Second (if this element is Non-Terminal) : we get its follow by these
    conditions:
        (if the element after it is Non-Terminal) :
            (if the first of the following Non-Terminal has epsilon): we
            add it to the vector of first in the current Non-Terminal
            struct and move to the next Non-Terminal.
                (if it has epsilon in its first) : repeat this step until
                (if a Non-Terminal comes with a first epsilon free) : add
                it to the first vector of the current Non-Terminal struct,
                stop and move to another Non-Terminal to find its followers.
                (if we reach to the end of the production) : we add the
                Non-Terminal key to the follower vector in the struct
                of the current Non-Terminal.
                (if we reach to Terminal) : we add it to the terminal vector
                in the current Non-Terminal struct.
        (if the element after it is Terminal) : add this Terminal to the
        vector of terminals in this Non-Terminal struct.
Fourth : Add a '$' to the Terminal vector of the start Non-Terminal struct.
```

```
void generateFollow():
----------------------
This method is responsible for generating the list of Terminal followers for
each Non-Terminal by following the map<string, vector<string> > follow, in
which the Non-Terminal as key and vector of its Terminal followers as value
First : we iterate on each Non-Terminal in the intial table we have filled
in the previous method.
    First : we iterate on the current Non-Terminal vector of followers,
        (if the boolean of the Non-Terminal required is false) : means that
        it hasnot yet calculated, so we skip it.
        (if the boolean of the Non-Terminal required is true) : we add its
        Terminals to the current Non-terminal value vector, then we remove
        this required Non-Terminal from the follower vector of the current
        Non-Terminal.
    Second : we iterate on the current Non-Terminal vector of first.
        we add the first Terminals to the current Non-terminal value vector,
        then, remove the added from the vector in the struct.
    Third : we iterate on the current Non-Terminal vector of Terminals.
        we add the Terminals to the current Non-terminal value vector,
        then, remove the added from the vector in the struct.
    Fourth : we check if the three vectors of the current Non-Terminal
        become empty, we set the coolean in the struct of the current
        Non-Terminal by true.
    Fifth : we And the boolean in the struct of the current Non-Terminal and
        we stop when this And result of all Non-Terminals is true.
```

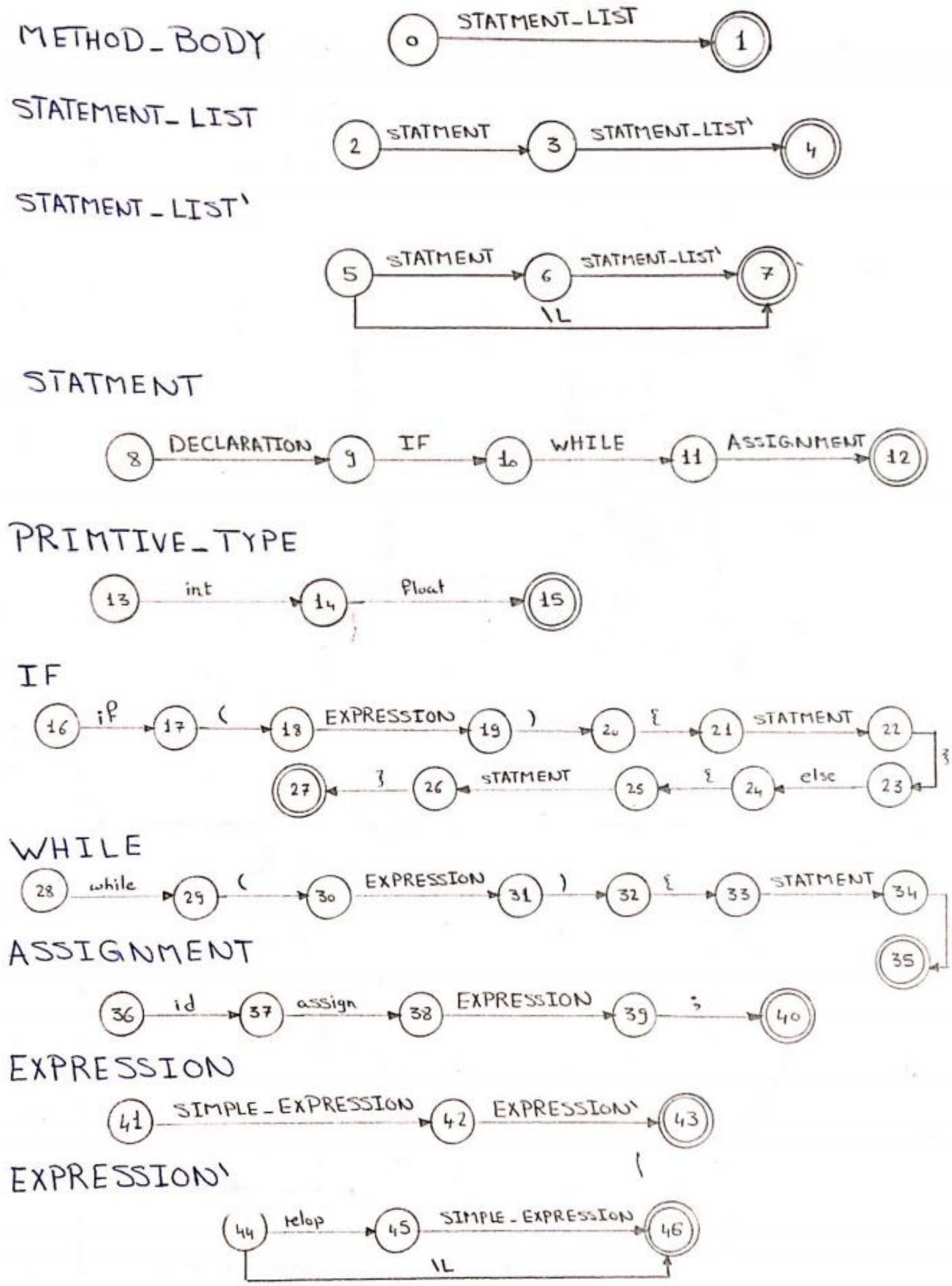
2.5 Table construction

```
construct_table() {
    // this function constructs the table by reading the grammar from and
       calculating the first and follew sets of each nonterminal.
    call readFile().
    call calc_firsts().
    call generateFollow().
    for each production rule in each nonterminal N then
        fill the table with SYNCH if the N has EPSILON in its first set.
        add the productions in the table slots.
}
```

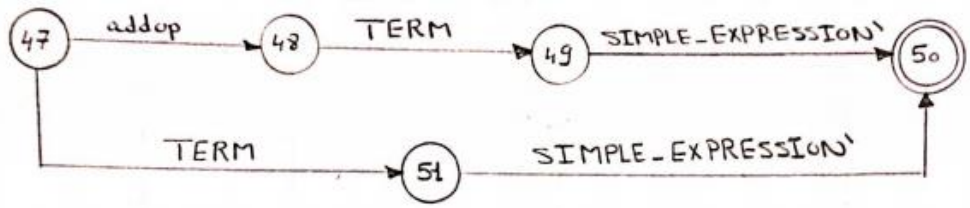
2.6 Parsing

```
parse() {
    call get_token.
    while flag is asserted
        if the token and the stack are not empty
            if top of stack was terminal
                if token is matching with stack top
                    pop from stack.
                    call get_token.
                else
                    specify suitable error.
                    call get_token or pop from stack dependes on the error
                    specified.
            else
                go to the table to get the production associated to the current
                token with non-terminal on the top of the stack.
                if entry was empty
                    specify error.
                    call get_token.
                else if entry was SYNCH
                    specify error.
                    pop from the stack.
                else
                    print the production to specify left most derivation.
                    pop form stack.
                    update stack (push the new production in reversed order).
        else
            specify suitable errors until matching between end of the input and
            end of stack.
            update flag.
}
```

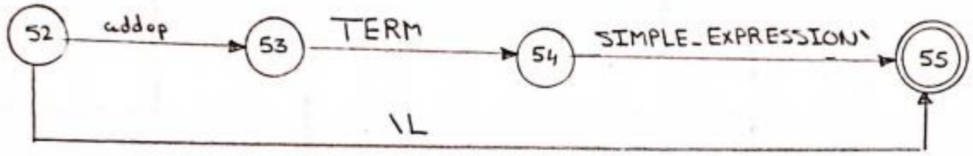
3. Transition diagram



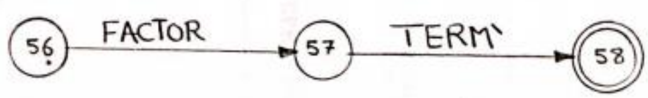
SIMPLE_EXPRESSION



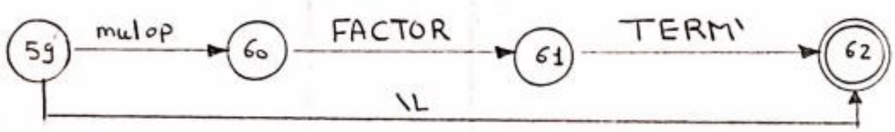
SIMPLE_EXPRESSION'



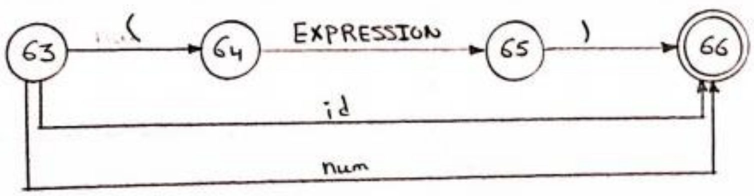
TERM



TERM'



FACTOR



4. Parsing tables

Predective Parsing Table

**** Mapping :**

Name	original	Name	original
A	METHOD_BODY	a	$
B	STATEMENT_LIST	b	(
C	STATEMENT	c	)
D	DECLARATION	d	;
E	PRIMITIVE_TYPE	e	addop
F	IF	f	assign
G	WHILE	g	else
H	ASSIGNMENT	h	float
I	EXPRESSION	i	id
J	EXPRESSION'	j	if
K	SIMPLE_EXPRESSION	k	int
L	TERM	l	mulop
M	FACTOR	m	num
N	STATEMENT_LIST'	n	relop
O	SIMPLE_EXPRESSION'	o	while
P	TERM'	p	{
Q	SYNCH	q	}

**** Table :**

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
A	Q							B	B	B	B				B		
B	Q							CN	CN	CN	CN				CN		
C	Q							D	H	F	D				G		Q
D	Q							Eid	Q	Q	Eid				Q		Q
E								h	Q		k						
F	Q							Q	Q	jbIcpCqgpCq	Q				Q		Q
G	Q							Q	Q	Q	Q				obIcpCq		Q
H	Q							Q	ifId	Q	Q				Q		Q
I		KJ	Q	Q	KJ				KJ				KJ				
J														nK			
K		LO	Q	Q	eLO				LO				LO	Q			
L		MP	Q	Q	Q				MP				MP	Q			
M		bIc	Q	Q	Q				i			Q	m	Q			
N								CN	CN	CN	CN				CN		
O					eLO												
P												lMP					

```
                         Leftmost Derivation
                         -------------------
METHOD_BODY -> STATEMENT_LIST
STATEMENT_LIST -> STATEMENT STATEMENT_LIST'
STATEMENT -> DECLARATION
DECLARATION -> PRIMITIVE_TYPE id ;
PRIMITIVE_TYPE -> int
STATEMENT_LIST' -> STATEMENT STATEMENT_LIST'
STATEMENT -> ASSIGNMENT
ASSIGNMENT -> id assign EXPRESSION ;
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION'
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSION'
TERM -> FACTOR TERM'
FACTOR -> num
TERM' -> \L
SIMPLE_EXPRESSION' -> \L
EXPRESSION' -> \L
STATEMENT_LIST' -> STATEMENT STATEMENT_LIST'
STATEMENT -> IF
IF -> if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION'
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSION'
TERM -> FACTOR TERM'
FACTOR -> id
TERM' -> \L
SIMPLE_EXPRESSION' -> \L
EXPRESSION' -> relop SIMPLE_EXPRESSION
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSION'
TERM -> FACTOR TERM'
FACTOR -> num
TERM' -> \L
SIMPLE_EXPRESSION' -> \L
STATEMENT -> ASSIGNMENT
ASSIGNMENT -> id assign EXPRESSION ;
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION'
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSION'
TERM -> FACTOR TERM'
FACTOR -> num
TERM' -> \L
SIMPLE_EXPRESSION' -> \L
EXPRESSION' -> \L
Missing 'else', inserted.
Missing '{', inserted.
Illegal 'STATEMENT', discarded.
Missing '}', inserted.
STATEMENT_LIST' -> \L
Parsing is done.
```

5. Assumptions

- Grammar lines format:

"# Non-terminal_Name = (any production)" or "| (any production)".

- Epsilon is a terminal we expect it as '\L'.
- Assume that if no more tokens then get_token() would return empty string (“”).
- In case of left factoring, it take one common factor each loop.
  - Example

    A → Ba | Bab

    Will be

    A → BA’

    A’ → aA’’

    A’’ → b | €