Top-Down Parsing

1. Given the grammar rule for an if-statement:

If-stmt → if (*exp*) *statement*
| if (*exp*) *statement* else *statement*

write pseudo-code to parse this grammar by recursive descent

Answer:

The EBNF of the if-statement

If-stmt → if (*exp*) *statement* [else *statement*]

Square brackets of the EBNF are translated into a test in the code for if-stmt.

```
procedure if-stmt;
    begin
      match( if );
      match( ( );
      exp;
      match( ) );
      statement;
      if token = else then
        match (else);
        statement;
      end if;
    end if-stmt;

procedure match( expectedToken);
begin
  if token = expectedToken then
    getToken;
  else
    error;
  end if;
end match
```

2. Consider the grammar

expr → expr addop term|term
addop → +|-
term → term mulop factor | factor
mulop →*
factor →(expr) | number

a) Remove the left recursion

exp → term exp’
exp’→ addop term exp’|ε

addop → + -
term → factor term'
term' → mulop factor term'|ε
mulop →*
factor →(expr) | number

b) Construct First and Follow sets for the non terminal of the resulting grammar
Write out each choice separately in order:

(1) exp → term exp'
(2) exp'→ addop term exp'
(3) exp'→ε
(4) addop → +
(5) addop →-
(6) term → factor term'
(7) term' → mulop factor term'
(8) term' → ε
(9) mulop →*
(10) factor →(expr)
(11) factor → number

First Set

1. Definition

- **Let X be a grammar symbol(a terminal or non-terminal) or ε. Then First(X) is a set of terminals or ε, which is defined as follows:**
  1) If X is a terminal or ε, then First(X) = {X};
  2) If X is a non-terminal, then for each production choice X→X1X2…Xn, First(X) contains First(X1)-{ε}.
  If also for some i<n, all the set First(X1)..First(Xi) contain ε,the first(X) contains First(Xi+1)-{ε}.
  3) IF all the set First(X1)..First(Xn) contain ε, the First(X) contains ε.
- **Let α be a string of terminals and non-terminals, X1X2…Xn. First(α) is defined as follows:**
  1) First(α) contains First(X1)-{ε};
  2) For each i=2,…,n, if for all k=1,..,i-1, First(Xk) contains ε, then First(α) contains First(Xk)-{ε}.
  3) IF all the set First(X1)..First(Xn) contain ε, the First(α) contains ε.

- **The computation process for above First Set**

Grammar Rule	Pass 1	Pass 2	Pass 3
exp → term exp'			First(exp)={(,number}
exp'→ addop term exp'		First(exp')={+, -, **ε** }	
exp'→ε	First(exp')={ **ε** }		
addop → +	First(addop)={+ }		
addop →-	First(addop)={+,-}		
term → factor term'		First(term)={(,number}	
term' → mulop factor term'		First(term')={ *, **ε** }	
term' → ε	First(term')={ **ε** }		
mulop →*	First(mulop)={*}		
factor →(expr)	First(factor)={(}		
factor → number	First(factor)={(,number}		

- The First sets are as follows:
  First(exp)={(,number)
  First(exp')={+,-, ε}
  First(term)={(,number)
  First(term')={*, ε}
  First(factor)={(,number}
  First(addop)={+,-}
  First(mulop)={*}

The Follow sets

1. Definition

Given a non-terminal A, the set Follow(A) is defined as follows.

(1) if A is the start symbol, the $ is in the Follow(A).

(2) if there is a production B→αAγ,then First(γ)-{ε} is in Follow(A).

(3) if there is a production B→αAγsuch that ε in First(γ), then Follow(A) contains Follow(B).

- **Note: The symbol $ is used to mark the end of the input.**
  - **The empty "pseudotoken" ε is never an element of a follow set.**
  - **Follow sets are defined only for non-terminal.**
  - **Follow sets work "on the right" in production while First sets work "on the left"in the production.**
- **Given a grammar rule A →αB, Follow(B) will contain Follow(A),**
  - **the opposite of the situation for first sets, if A →Bα,First(A) contains First(B),except possibly for ε.**
- **According to the definition of Follow sets**
  Follow(exp)= { $} union{)}
  Follow(exp')= Follw(exp)
  Fllow(term)= (First(exp') **-{ε})** union Follow(exp) union Follow(exp')
  Follow(term')= Follow(term)

Follow(factor)= (First(term’) **-{ε})** union Follow(term) union Follow(term’)
Follow(addop)= First(term)
Follow(mulop)= First(factor)

- The progress of above computation

Grammar Rule	Pass 1	Pass 2
expr → term exp’	Follow(term)=First(exrp’)-{ **ε** }={ +,-}	Follow(term)= { +,-}union Follow(expr)={+,-,$,)} Follow(expr’)= Follow(expr)={$,)}
expr’→ addop term expr’	Follow(addop)= First(term)= {(, number}	Follow(term)= {+,-,$,)}union Follow(expr’)= {+,-,$,)}
exp’→ε		
addop → +		
addop →-		
term → factor term’	Follow(factor)= (First(term’)-{ **ε** })union Follow(term)={ ***,+,-** } Follow(term’)= Follow(term)={ +, - }	Follow(factor)={ ***,+,-** }union Follow(term)= {*,+,-,$,)} Follow(term’)= { +, - }union Follow(term)= { +,-,$,)}
term’ → mulop factor term’	Follow(mulop)= First(factor)= {(, number} Follow(facor)={ *,+,-} union Follow(term’)= { *, +,-}	Follow(facor)={ *, +,-}union Follow(term’)= { *, +,-,$,)}
term’ → ε		
mulop →*		
factor →(expr)	Follow(expr)= {$}union{) }={$,)}	
factor → number		

- The Follow sets are as follows:
Follow(exp)={$,) }
Follow(exp’)={$,)}
Follow(addop)={(,number)
Follow(term)={ $,+,-,)}
Follow(term’)={$,+,-,)}
Follow(mulop)={(,number}
Follow(factor)={ $,+,-, *,)}

c) Construct LL(1) parsing table for the resulting grammar

The first and follow set

First Sets	Follow Sets
First(exp)={(,number)	**Follow(exp)={$,) }**
First(exp')={+,-, ε}	**Follow(exp')={$,)}**
First(term)={(,number)	**Follow(term)={ $,+,-,)}**
First(term')={*, ε}	**Follow(term')={$,+,-,)}**
First(factor)={(,number}	**(factor)={ $,+,-, *,)}**
First(addop)={+,-}	**Follow(addop)={(,number)**
First(mulop)={*}	**Follow(mulop)={(,number}**

the LL(1) parsing table

M[N,T]	(	number	)	+	-	*	$
Exp	**exp → term exp'**	**exp → term exp'**					
Exp'			**exp'→ε**	**exp'→ addop term exp'**	**exp'→ addop term exp'**		**exp'→ε**
Addop				**addop → +**	**addop → -**		
Term	**term → factor term'**	**term → factor term'**					
Term'			**term' →ε**	**term' →ε**	**term' →ε**	**term' → mulop factor term'**	**term' →ε**
Mulop						**mulop →***	
factor	**factor →(expr)**	**factor → number**					

<u>Note:</u>

* to fill in the row Exp, you search for the rules where exp is at left side, there is only one rule

(1) exp → term exp'

So the columns where this production is inserted are the columns that correspond to the tokens in First(term), so the columns are : (, number .

* to fill in the row Exp’, you search for the rules where exp’ is at left side, there are two rules

(2) exp’→ addop term exp’
(3) exp’→ε

- From rule #2 the columns where this production is inserted are the columns that correspond to the tokens in First(addop), so the columns are : + , -.

- From rule #3 the columns where this production is inserted are the columns that correspond to the tokens in Follow(exp’), so the columns are : $,).

d) Show the actions of the corresponding LL(1) parser, given the input string 3+4

(1) exp → term exp’
(2) exp’→ addop term exp’
(3) exp’→ε
(4) addop → +
(5) addop →-
(6) term → factor term’
(7) term’ → mulop factor term’
(8) term’ → ε
(9) mulop →*
(10) factor →(expr)
(11) factor → number

Step	Parsing Stack	Input	Action
1	S $	3+4 $	(1)exp → term exp’
2	term exp’ $	3+4 $	(6) term → factor term’
3	factor term’exp’ $	3+4 $	(11) factor → number
4	number term’exp’ $	3+4 $	Match
5	term’exp’ $	+4 $	(8) term’ → ε
6	exp’ $	+4 $	(2) exp’→ addop term exp’
7	addop term exp’$	+4 $	(4) addop → +
8	+ term exp’$	+4 $	Match
9	term exp’$	4 $	(6) term → factor term’
10	factor term’ exp’$	4 $	(11) factor → number
11	number term’ exp’$	4 $	Match
12	term’ exp’$	$	(8) term’ → ε
13	exp’$	$	(3) exp’→ε
14	$	$	accept

<u>**Note**</u>

* in the first step we insert the start symbol (exp) in the stack
* The action in each step is selected from the parsing table based on the non terminal on the top of the stack, and the current token from input string:

For example:

- in step # 2 the action is the production in the parsing table cell M[term, number] which is

(6) term → factor term’
- in step # 5 the action is the production in the parsing table cell M[term’, +] which is
  (8) term’ → ε
- If the top of the stack is a terminal, in this case the action is Match.

3. Consider the following grammar

Statement → if-stmt | other
If-stmt → if (exp) *statement*
| if (exp) *statement* else *statement*
Exp → 0 | 1

a) Left factor this grammar

Statement → if-stmt | other
If-stmt → if (exp) statement else-part
Else-part → else statement | ε
Exp → 0 | 1

b) Construct First and Follow sets for the non terminal of the resulting grammar

- **We write out the grammar rule choice separately and number them:**
  (1) Statement → if-stmt
  (2) Statement → other
  (3) If-stmt → if (exp) statement else-part
  (4) Else-part → else statement
  (5) Else-part →ε
  (6) Exp → 0
  (7) Exp → 1
- Note: This grammar does have an ε-production, but the only nullable non-terminal *else-part* will not in the beginning of left side of any rule choice and will not complicate the computation process.
- The First Sets:
  First(statement)={if,other}
  First(if-stmt)={if}
  First(else-part)={else,ε}
  First(exp)={0,1}

The computation process for above First Sets

Grammar Rule	Pass 1	Pass 2
Statement → if-stmt		**First(statement)={if,other}**
Statement → other	**First(statement)={other}**	
If-stmt → if (exp) statement else-part	**First(if-stmt)={if}**	
Else-part → else statement	**First(else-part)={else}**	
Else-part →ε	**First(else-part)={else,ε}**	

Exp → 0	**First(exp)={1}**	
Exp → 1	**First(exp)={0,1}**	

- The Follow Sets:

  Follow(statement)={\$,else}
  Follow(if-statement)={\$,else}
  Follow(else-part)={\$,else}
  Follow(exp)={) }

The computation process for above Follow Sets

- The first and follow set

First Sets	Follow Sets
First(statement)={if,other} **First(if-stmt)={if}** **First(else-part)={else,ε}** **First(exp)={0,1}**	**Follow(statement)={\$,else}** **Follow(if-statement)={\$,else}** **Follow(else-part)={\$,else}** **Follow(exp)={) }**

c) Construct the LL(1) parsing table for the resulting
the LL(1) parsing table

M[N,T]	**If**	**Other**	**Else**	**0**	**1**	**\$**
Statement	**Statement → if-stmt**	**Statement → other**				
If-stmt	**If-stmt → if (exp) statement else-part**					
Else-part			**Else-part → else statement Else-part →ε**			**Else-part →ε**
exp				**Exp → 0**	**Exp → 1**	

Notice for Example: If-Statement
A grammar is an LL(1) grammar if the associated LL(1) parsing table has at most on production in each table entry

- **The entry M[else-part, else] contains two entries, i.e.** ***the dangling else ambiguity.***
- **Disambiguating rule:** ***always prefer the rule that generates the current look-ahead token over any other, and thus the production***

  Else-part → else statement
  or
  Else-part →ε
- **With this modification, the above table will become unambiguous**
  - **The grammar can be parsed as if it were an LL(1) grammar**

d) Show the action of corresponding LL(1) parser given the input string **If (0) if (1) other else other**

- **The parsing actions for the string:**

If (0) if (1) other else other

- **(for conciseness, statement= S, if-stmt=I, else-part=L, exp=E, if=i, else=e, other=o)**

Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
			Match
			Match
			S→I
			I→i(E)SL
			Match
			Match
			E→1
			Match
			match
			S→o
			match
			L→eS
			Match
			S→o
			match
			L→ε
22	$	$	accept

4. Consider the following grammar

Stmt-sequence→stmt; stmt-sequence | stmt
Stmt→s

a) Left factor this grammar

Stmt-sequence→stmt stmt-seq’
Stmt-seq’→; stmt-sequence | ε

b) Construct First and Follow sets for the non terminal of the resulting grammar

The first and follow set

First Sets	Follow Sets
First(stmt-sequence)={s} **First(stmt)={s}** **First(stmt-seq’)={;, ε}**	**Follow(stmt-sequence)={$}** **Follow(stmt)={;}** **Follow(stmt-seq’)={$}**

c) Construct the LL(1) parsing table for the resulting

the LL(1) parsing table

M[N,T]	S	;	$
Stmt-sequence	**Stmt-sequence →stmt stmt-seq'**		
Stmt	**stmt→s**		
Stmt-seq'		**Stmt-seq' →; stmt-sequence**	**Stmt-seq' →ε**

d) Show the action of corresponding LL(1) parser given the input string **s; s**

----

5. Given the grammar

exp → exp addop term|term
addop → + |-
term → term mulop factor | factor
*mulop →**
*factor →(exp)| **number***

Write pseudo-code to parse this grammar by recursive descent

Answer:

- The corresponding EBNF is

exp → term { addop term }
***addop→* + | -**
term → factor { mulop factor }
mulop→ *
factor →* (*exp*) | *numberr

```
procedure exp;
    begin
      term;
      while token = + or token = - do
            match(token);
            term;
      end while;
    end exp;

procedure term;
    begin
      factor;
      while token = * do
            match(token);
            factor;
      end while;
    end exp;

procedure factor
begin
```

```
case token of
( :  match( ( );
 exp;
match( ));
  number:
    match (number);
  else error;
  end case;
end factor

procedure match( expectedToken);
begin
  if token = expectedToken then
    getToken;
  else
    error;
  end if;
end match
```

6. Consider the following grammar

$S \rightarrow (S)\ S | \varepsilon$

a) Construct First and Follow sets for the non terminals of the grammar

----

b) Construct the LL(1) parsing table for the resulting

M[N,T]	**(**	**)**	**$**
S	**S→(S) S**	**S→ε**	**S→ε**

c) Show the action of corresponding LL(1) parser given the input string ( )

Steps	Parsing Stack	Input	Action
1	$S	( ) $	S→(S) S
2	$S)S(	( ) $	match

3	$S)S	)$	S→ε
4	$S)	)$	match
5	$S	$	S→ε
6	$	$	accept

7. Left factor the following grammar:
   exp → term+exp |term

   Answer:
   exp → term exp’
   exp’→ + exp|ε