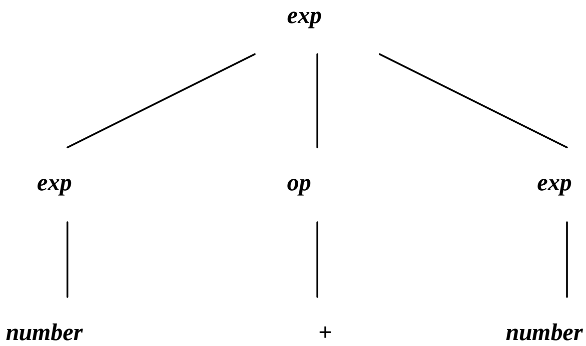
CHAPTER 4: TOP-DOWN PARSING

1. Introduction

1.1. Concept of Top-Down Parsing (1)

- **It parses an input string of tokens by *tracing out the steps* in a leftmost derivation.**
  - **And the implied traversal of the parse tree is a preorder traversal and, thus, occurs from the root to the leaves.**
- **The example:**
  - **number + number, and corresponds to the parse tree**



- **The above parse tree corresponds to the leftmost derivations:**

(1) ***exp => exp op exp***
(2) ***=> number op exp***
(3) ***=> number + exp***
(4) ***=> number + number***

1.2. Two forms of Top-Down Parsers

- ***Predictive parsers*:**
  - **attempts to predict the next construction in the input string using one or more look-ahead tokens**
- ***Backtracking parsers*:**
  - **try different possibilities for a parse of the input, backing up an arbitrary amount in the input if one possibility fails.**
  - **It is more powerful but much slower, unsuitable for practical compilers.**

1.3. Two kinds of Top-Down parsing algorithms

- ***Recursive-descent parsing*:**
  - **is quite versatile and suitable for a handwritten parser.**
- ***LL(1) parsing*:**
  - **The first “L” refers to the fact that it processes the input from left to right;**
  - **The second “L” refers to the fact that it traces out a leftmost derivation for the input string;**
  - **The number “1” means that it uses only one symbol of input to predict the direction of the parse.**

4.1 Top-Down Parsing by Recursive-Descent

4.1.1 The Basic Method of Recursive-Descent

1. The idea of Recursive-Descent Parsing

- **Viewing the grammar rule for a non-terminal A as a definition for a procedure to recognize an A**
- **The right-hand side of the grammar for A specifies the structure of the code for this procedure**

- **Requiring the Use of EBNF**

- **The Expression Grammar:**
  exp* → *term { addop term }
  ***addop* → + | -**
  term* → *factor { mulop factor }
  ***mulop* → ***
  factor* → (*exp*) | *numberr

a) A recursive-descent procedure that recognizes a *factor*

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

- **The token keeps the current next token in the input (one symbol of look-ahead)**

- **The Match procedure matches the current next token with its parameters, advances the input if it succeeds, and declares error if it does not**

b) Match Procedure

- **The Match procedure matches the current next token with its parameters,**
  - **advances the input if it succeeds, and declares error if it does not**

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

c)

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

d)

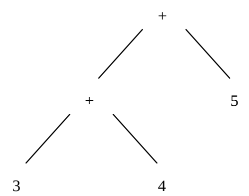
```
procedure term;
begin
  factor;
```

```
    while token = * do
            match(token);
            factor;
    end while;
  end term;
```

Some Notes

- **The method of turning grammar rule in EBNF into code is quite powerful.**
- **There are a few pitfalls, and care must be taken in scheduling the actions within the code.**
- **In the previous pseudo-code for exp:**
  (1) The match of operation should be before repeated calls to term;
  (2) The global token variable must be set before the parse begins;
  (3) The getToken must be called just after a successful test of a token

2. Construction of the syntax tree
The Expression 3+4+5



a) The pseudo-code for constructing the syntax tree(1)

```
function exp : syntaxTree;
     var temp, newtemp: syntaxTree;
     begin
       temp:=term;
       while token=+ or token = - do
                    case token of
                    + : match(+);
                    newtemp:=makeOpNode(+);
                    leftChild(newtemp):=temp;
                    rightChild(newtemp):=term;
                    temp=newtemp;
```

b) The pseudo-code for constructing the syntax tree(2)

```
                -:match(-);
                newtemp:=makeOpNode(-);
                leftChild(newtemp):=temp;
                rightChild(newtemp):=term;
                temp=newtemp;
end case;
          end while;
          return temp;
end exp;
```

c) A simpler one

```
function exp : syntaxTree;
     var temp, newtemp: syntaxTree;
     begin
       temp:=term;
```

```
        while token=+ or token = - do
                    newtemp:=makeOpNode(token);
                    match(token);
                    leftChild(newtemp):=temp;
                    rightChild(newtemp):=term;
                    temp=newtemp;
          end while;
          return temp;
end exp;
```

4.1.3 Further Decision Problems

More formal methods to deal with complex situation

(1) It may be difficult to convert a grammar in BNF into EBNF form;

(2) It is difficult to decide when to use the choice $A \rightarrow \alpha$ and the choice $A \rightarrow \beta$; if both α and β begin with non-terminals. Such a decision problem requires the computation of the First Sets.

(3) It may be necessary to know what token legally coming after the non-terminal A, in writing the code for an ε-production: $A \rightarrow \varepsilon$. Such tokens indicate A may disappear at this point in the parse. This set is called the Follow Set of A.

(4) It requires computing the First and Follow sets in order to detect the errors as early as possible. Such as ")3-2)", the parse will descend from exp to term to factor before an error is reported.

4.2 LL(1) Parsing

4.2.1 The Basic Method of LL(1) Parsing

1. Main idea

- **LL(1) Parsing uses an explicit stack rather than recursive calls to perform a parse**
- **An example:**
  - **a simple grammar for the strings of balanced parentheses:**

    S→(S) S|ε
- **The following table shows the actions of a top-down parser given this grammar and the string ( )**

Table of Actions

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

2. General Schematic

- **A top-down parser begins by pushing the start symbol onto the stack**
- **It accepts an input string if, after a series of actions, the stack and the input become empty**
- **A general schematic for a successful top-down parse:**

```
$ StartSymbol      Inputstring$
  …                    …     //one of the two actions
  …                    …     //one of the two actions
$                    $  accept
```

a) Two Actions

- **The two actions**

  (1) Generate: Replace a non-terminal A at the top of the stack by a string α(in reverse) using a grammar rule A →α, and

  (2) Match: Match a token on top of the stack with the next input token.
- **The list of generating actions in the above table:**

  S => (S)S [S→(S) S]

  => ( )S [S→ε]

  => ( ) [S→ε]
- **Which corresponds precisely to the steps in a leftmost derivation of string ( ).**
- **This is the characteristic of top-down parsing.**

4.2.2 The LL(1) Parsing Table and Algorithm

1. Purpose and Example of LL(1) Parsing Table

- **Purpose of the LL(1) Parsing Table:**
  - **To express the possible rule choices for a non-terminal A when the A is at the top of parsing stack based on the current input token (the look-ahead).**
- **The LL(1) Parsing table for the following simple grammar:**

S→(S) S|ε

M[N,T]	(	)	$
S	S→(S) S	S→ε	S→ε

2. The General Definition of Table

- **The table is a two-dimensional array indexed by non-terminals and terminals**
- **Containing production choices to use at the appropriate parsing step called M[N,T]**
  - **N is the set of non-terminals of the grammar**
  - **T is the set of terminals or tokens (including $)**
- **Any entrances remaining empty**
  - **Representing potential errors**

3. Table-Constructing Rule

- **The table-constructing rule**
  - **If A→α is a production choice, and there is a derivation α=>*aβ, where a is a token, then add A→αto the table entry M[A,a];**
  - **If A→α is a production choice, and there are derivations α=>*ε and S$=>*βAaγ, where S is the start symbol and a is a token (or $), then add A→αto the table entry M[A,a];**

4. A Table-Constructing Case

- **The constructing-process of the following table**
  - **For the production : S→(S) S, α=(S)S, where a=(, this choice will be added to the entry M[S, (] ;**
  - **Since: S=>(S)Sε , rule 2 applied withα= ε, β=(,A = S, *a* =), and γ=S$,so add the choice S→εto M[S,)]**
  - **Since S$=>* S$, S→εis also added to M[S, $].**

M[N,T]	(	)	$
S	S→ (S) S	S→ ε	S→ ε

5. Properties of LL(1) Grammar

- **Definition of LL(1) Grammar**
  - **A grammar is an LL(1) grammar if the associated LL(1) parsing table has at most on production in each table entry**
- **An LL(1) grammar cannot be ambiguous**

6. A Parsing Algorithm Using the LL(1) Parsing Table
(* assumes $ marks the bottom of the stack and the end of the input *)

push the start symbol onto the top the parsing stack;
while the top of the parsing stack ≠ $ and
the next input token ≠ $ do
if ***the top of the parsing stack is terminal a and the next input token = a***
then (* match *)
pop the parsing stack;
advance the input;

else if ***the top of the parsing stack is non-terminal A***
and ***the next input token is terminal a***
and ***parsing table entry M[A,a] contains production A→ X1X2…Xn***
then (* generate *)
pop the parsing stack;
for i:=n downto 1 do
push Xi onto the parsing stack;
else error;
if ***the top of the parsing stack = $***
and the next input token = $
then accept
else error.

4.2.3 Left Recursion Removal and Left Factoring

1. Repetition and Choice Problem

- **Repetition and choice in LL(1) parsing suffer from similar problems to be those that occur in recursive-descent parsing**
  - **and for that reason we have not yet been able to give an LL(1) parsing table for the simple arithmetic expression grammar of previous sections.**
- **Solve these problems for recursive-descent by using EBNF notation**
  - **We cannot apply the same ideas to LL(1) parsing;**
  - **instead, we must rewrite the grammar within the BNF notation into a form that the LL(1) parsing algorithm can accept.**

2. Two standard techniques for Repetition and Choice

- **Left Recursion removal**

  exp → exp addop term | term

  (in recursive-descent parsing, EBNF: exp→ term {addop term})
- **Left Factoring**

  If-stmt → if (exp) statement

  | if (exp) statement else statement

  (in recursive-descent parsing, EBNF:

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

3. Left Recursion Removal

- **Left recursion is commonly used to make operations left associative, as in the simple expression grammar, where**

  exp → exp addop term | term
- **Immediate left recursion:**
  The left recursion occurs only within the production of a single non-terminal.

  exp → exp + term | exp - term |term
- **Indirect left recursion:**

Never occur in actual programming language grammars, but be included for completeness.

A → Bb |…

B → Aa |…

a) CASE 1: Simple Immediate Left Recursion

- **A → Aα| β**

  Where, αandβare strings of terminals and non-terminals;

  βdoes not begin with A.
- **The grammar will generate the strings of the form:** $\beta\alpha^{n}$
- ***We rewrite this grammar rule into two rules:***
  A → βA'
  To generate β first;
  A' → αA'| ε
  To generate the repetitions of α, using right recursion.

<u>Example</u>

- **exp → exp addop term | term**

- **To rewrite this grammar to remove left recursion, we obtain**

exp → term exp’

exp’ → addop term exp’ | ε

b) CASE2: General Immediate Left Recursion

A → Aα1| Aα2| … |Aαn|β1|β2|…|βm

Where none of β1,…,βm begin with A.

The solution is similar to the simple case:

A →β1A’|β2A’| …|βmA’

A’ → α1A’| α2A’| … |αnA’|ε

Example

- **exp → exp + term | exp - term |term**

- **Remove the left recursion as follows:**

  exp → term exp’

  exp’ → + term exp’ | - term exp’ |ε

c) Notice

- **Left recursion removal not changes the language, but**
  - **Change the grammar and the parse tree**
- **This change causes a complication for the parser**

Example:

Simple arithmetic expression grammar	After removal of the left recursion
expr → expr addop term| term addop → +|- term → term mulop factor | factor mulop → * factor → (expr) | number	exp → term exp’ exp’→ addop term exp’| ε addop → + - term → factor term’ term’ → mulop factor term’| ε mulop → * factor → (expr) | number

d) Left-Recursion Removed Grammar and its Procedures

- **The grammar with its left recursion removed, exp and exp’ as follows:**

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

```
Procedure exp
   Begin
     Term;
     Exp';
   End exp;
```

```
Procedure exp'
   Begin
     Case token of
      +: match(+);
        term;
        exp';
      -: match(-);
        term;
        exp';
     end case;
   end exp'
```

4. Left Factoring

- **Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule**

  $A \to \alpha\beta|\alpha\gamma$
- **An LL(1) parser cannot distinguish between the production choices in such a situation**
- **The solution in this simple case is to "factor" the α out on the left and rewrite the rule as two rules:**

  $A \to \alpha A'$

  $A' \to \beta|\gamma$

a) Algorithm for Left Factoring a Grammar

While there are changes to the grammar do

For each non-terminal A do

Let α be a prefix of maximal length that is shared

By two or more production choices for A

If $\alpha \neq \varepsilon$ then

Let $A \to \alpha1|\alpha2|\ldots|\alpha n$ be all the production choices for A

And suppose that $\alpha1,\alpha2,\ldots,\alpha k$ share α, so that

$A \to \alpha\beta1|\alpha\beta2|\ldots|\alpha\beta k|\alpha K+1|\ldots|\alpha n$, the βj's share

No common prefix, and $\alpha K+1,\ldots,\alpha n$ do not share α

Replace the rule $A \to \alpha1|\alpha2|\ldots|\alpha n$ by the rules

$A \to \alpha A'|\alpha K+1|\ldots|\alpha n$

$A' \to \beta1|\beta2|\ldots|\beta k$

b) Example 4.5

- **Consider the following grammar for if-statements:**

  If-stmt → if (exp) statement

  | if (exp) statement else statement
- **The left factored form of this grammar is:**

  If-stmt → if (exp) statement else-part

  Else-part → else statement | ε

4.3 First and Follow Sets

The LL(1) parsing table construction involves the First and Follow sets

4.3.1 First Sets
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, α= 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.

2. Algorithm Computing First (A)

- ***Algorithm for computing First(A) for all non-terminal A:***

```
For all non-terminal A do First(A):={ };
While there are changes to any First(A) do
  For each production choice A→X1X2…Xn do
    K:=1; Continue:=true;
    While Continue= true and k<=n do
      Add First(Xk)-{ε} to First(A);
      If ε is not in First(Xk) then Continue:= false;
      k:=k+1;
    If Continue = true then add ε to First(A);
```

- ***Simplified algorithm in the absence of ε-production.***

```
For all non-terminal A do First(A):={ };
While there are changes to any First(A) do
  For each production choice A→X1X2…Xn do
      Add First(X1) to First(A);
```

3. Example

a)

- Simple integer expression grammar

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

- Write out each choice separately in order:

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

b) The computation process for above First Set

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

c) First Set for Above Example

- **We can use the simplified algorithm as there exists no ε-production**
- **The First sets are as follows:**

First(exp)={(,number}
First(term)={(,number}
First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

4.3.2 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.**

2. Algorithm for the computation of follow sets

- **Follow(start-symbol):={$};**
- **For all non-terminals A≠start-symbol do follow(A):={ };**
- **While there changes to any follow sets do**

  For each production A→X1X2…Xn do

  For each Xi that is a non-terminal do

  Add First(Xi+1Xi+2…Xn) – {ε} to Follow(Xi)

  If ε is in First(Xi+1Xi+2…Xn) then

  Add Follow(A) to Follow(Xi)

3. Example

a) The simple expression grammar.

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

b) The first sets:

First(exp)={(,number}
First(term)={(,number}
First(factor)={(,number}
First(addop)={+,-}
First(mulop)={*}

c)The progress of above computation

Grammar rule	Pass 1	Pass 2
exp → exp addop term	Follow(exp)={$,+,- } Follow(addop)={(,number} Follow(term)={ $,+,-}	Follow(term)={ $,+,-, *,) }
Exp → term		
term → term mulop factor	Follow(term)={ $,+,-, *} Follow(mulop)={(,number} Follow(factor)={ $,+,-, *}	Follow(factor)={ $,+,-, *,) }
term →factor		
factor →(exp)	Follow(exp)={$,+,-,) }	

d)The Follow sets:

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

4.3.3 Constructing LL(1) Parsing Tables

1. The table-constructing rules

(1) If A→α is a production choice, and there is a derivation α=>*aβ, where a is a token, then add A→α to the table entry M[A,a]

(2) If A→α is a production choice, and there are derivations α=>*εand S$=>*βAaγ, where S is the start symbol and a is a token (or $), then add A→α to the table entry M[A,a]

- **Clearly, the token a in the rule (1) is in First(α), and the token a of the rule (2) is in Follow(A).**
- **Thus we can obtain the following algorithmic construction of the LL(1) parsing table:**

2. Algorithm

- **Repeat the following two steps for each non-terminal A and production choice A→α.**
  - **For each token a in First(α), add A→α to the entry M[A,a].**
  - **If ε is in First(α), for each element a of Follow(A) (a token or $), add A→α to M[A,a].**

3. Example

a) The simple expression grammar.

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

b) The first and follow set

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

c) 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					

4.3.4 Extending the look ahead: LL(k) Parsers
Definition of LL(k)

- **The LL(1) parsing method can be extend to k symbols of look-ahead.**
- **Definitions:**
  - **First k(α)={*w*k | α=>* *w*}, where, *w*k is the first k tokens of the string *w* if the length of w > k, otherwise it is the same as *w*.**
  - **Follow k(A)={*w*k | S$=>*αA*w*}, where, *w*k is the first k tokens of the string *w* if the length of w > k, otherwise it is the same as *w*.**
- **LL(k) parsing table:**
  - **The construction can be performed as that of LL(1).**