Compiler Principle and Technology

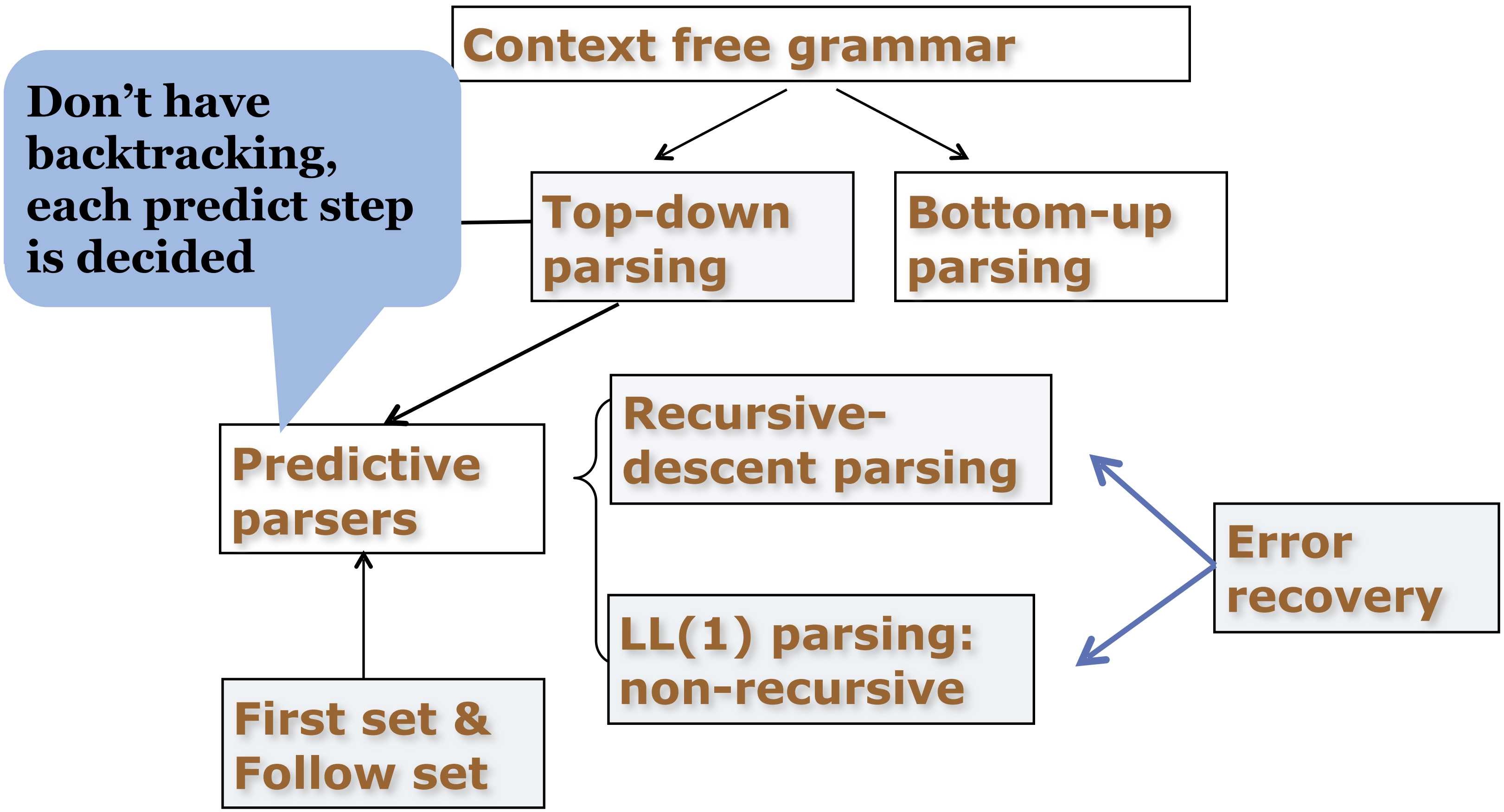
Prof. Dongming LU
Mar. 25th, 2015

4. Top-Down Parsing

PART ONE

Contents

Basic Concepts



4.1 Top-Down Parsing by Recursive-Descent

4.1.1 The Basic Method of Recursive-Descent

The idea of Recursive-Descent Parsing

- The grammar rule for a non-terminal A : a definition for a procedure to recognize an A
- The right-hand side of the grammar for A : the structure of the code for this procedure
- The Expression Grammar:
  - *exp* → *exp addop term*|*term*
  - *addop* → + |-
  - *term* → *term mulop factor* | *factor*
  - *mulop* →*
  - *factor* →(*exp*) | **number**

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**

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
```

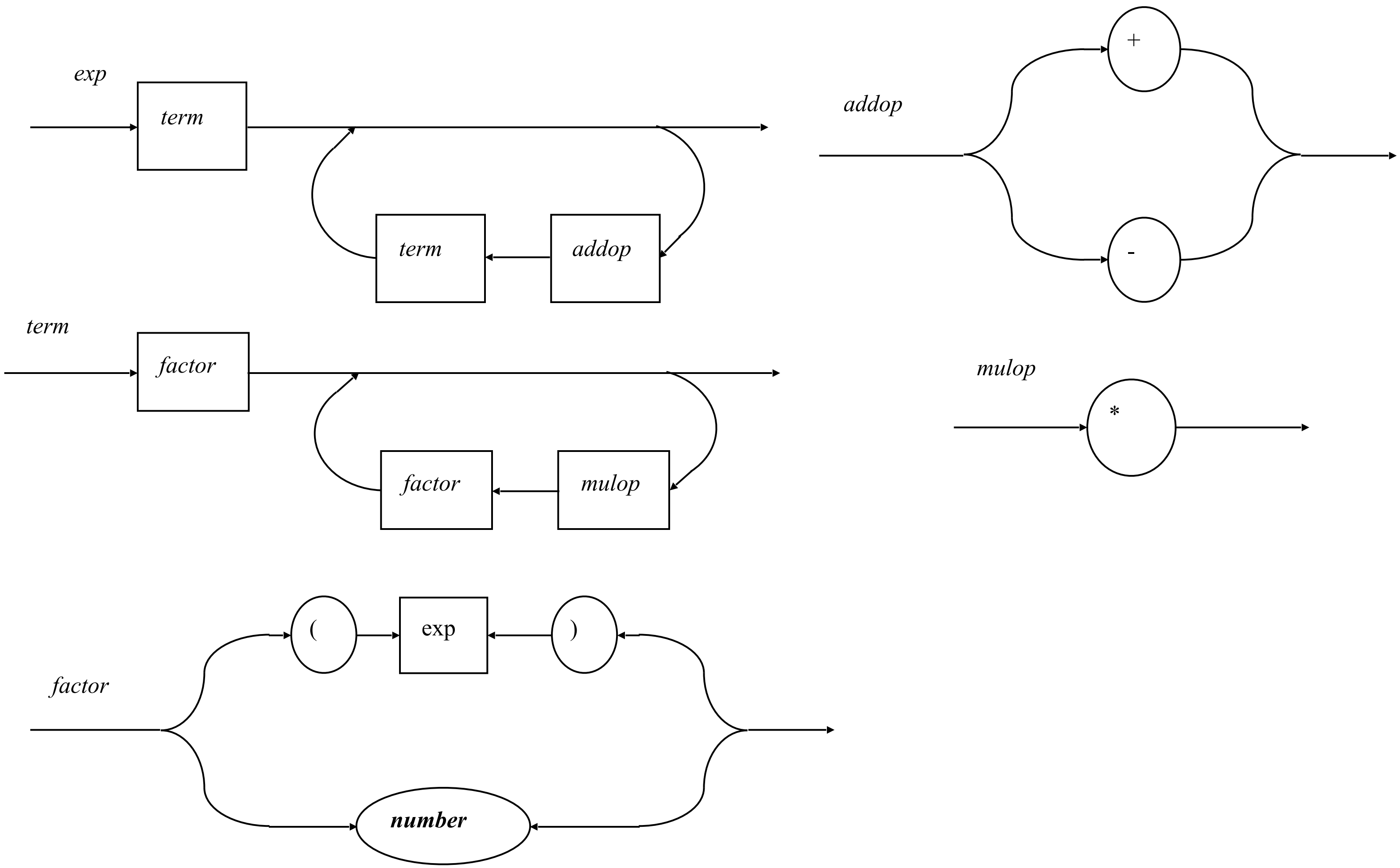
Requiring the Use of EBNF

- The corresponding EBNF is

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

- Writing recursive-decent procedure for the remaining rules in the expression grammar is not as easy for factor

The corresponding syntax diagrams



4.1.2 Repetition and Choice: Using EBNF

An Example

procedure ifstmt;
  begin
   match(**if**);
   match(**(**);
   exp;
   match(**)**);
   statement;
   if token = else then
    match (else);
    statement;
   end if;
  end ifstmt;

- The grammar rule for an if-statement:

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

Issuse

- Could not immediately distinguish the two choices because the both start with the token ***if***
- Put off the decision until we see the token **else** in the input

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*:

```
if token = else then
    match (else);
   statement;
end if;
```

- **Notes**
  - ➢ EBNF notation is designed to mirror closely the actual code of a recursive-descent parser,
  - ➢ So a grammar should always be translated into EBNF if recursive-descent is to be used.
  - ➢ It is natural to write a parser that matches each else token as soon as it is encountered in the input

EBNF for Simple Arithmetic Grammar(1)

The EBNF rule for :

$exp \rightarrow exp$ addop $term \mid term$

$exp \rightarrow term \{addop\ term\}$

The curly bracket expressing repetition can be translated into the code for a loop:

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

EBNF for Simple Arithmetic Grammar(2)

- The EBNF rule for term:
  term → factor {mulop factor}

  Becomes the code

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

Left associatively implied by the curly bracket

- The left associatively implied by the curly bracket (and explicit in the original BNF) can still be maintained within this code

```
function exp: integer;
   var temp: integer;
   begin
   temp:=term;
   while token=+ or token = -
     do
     case token of
     + : match(+);
         temp:=temp+term;
     -:  match(-);
             temp:=temp-term;
         end case;
     end while;
   return temp;
end exp;
```

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:

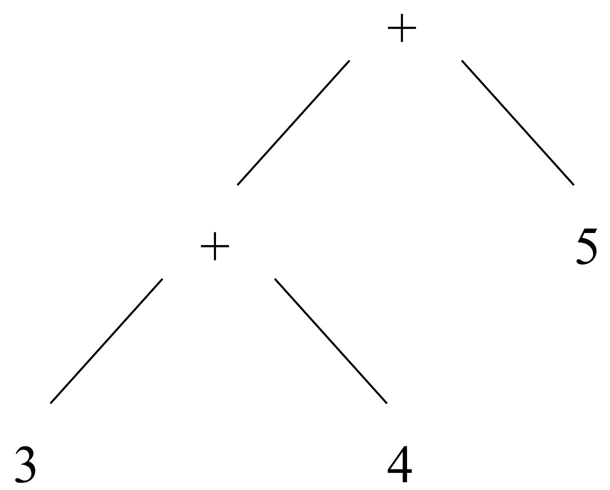
  (1) The match of operation should be before repeated calls to term;

  (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

Construction of the syntax tree

The expression: 3+4+5



The pseudo-code for constructing the syntax tree

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

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;
```

The pseudo-code for the if-statement procedure

```
function ifstatement: syntaxTree;
  var temp:syntaxTree;
  begin
    match(if);
    match(();
    temp:= makeStmtNode(if);
    testChild(temp):=exp;
    match());
    thenChild(temp):=statement;
  if token= else then
      match(else);
      elseChild(temp):=statement;
    else
      ElseChild(temp):=nil;
    end if;
  end ifstatement
```

4.1.3 Further Decision Problems

Characteristics of recursive-descent

The recursive-descent method simply translates the grammars into procedures, thus, it is very easy to write and understand, however, it is ad-hoc, and has the following drawbacks:

(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 $A \to \alpha$ and the choice $A \to \beta$; if both α and β begin with non-terminals. (requires the computation of the **First Sets**)

Characteristics of recursive-descent

(3) It may be necessary to know what token legally coming from the non-terminal A.

In writing the code for an ε-production: A →ε. Such tokens indicate. A may disappear at this set is called the **Follow Set** of A.

We need a more general and formal method !

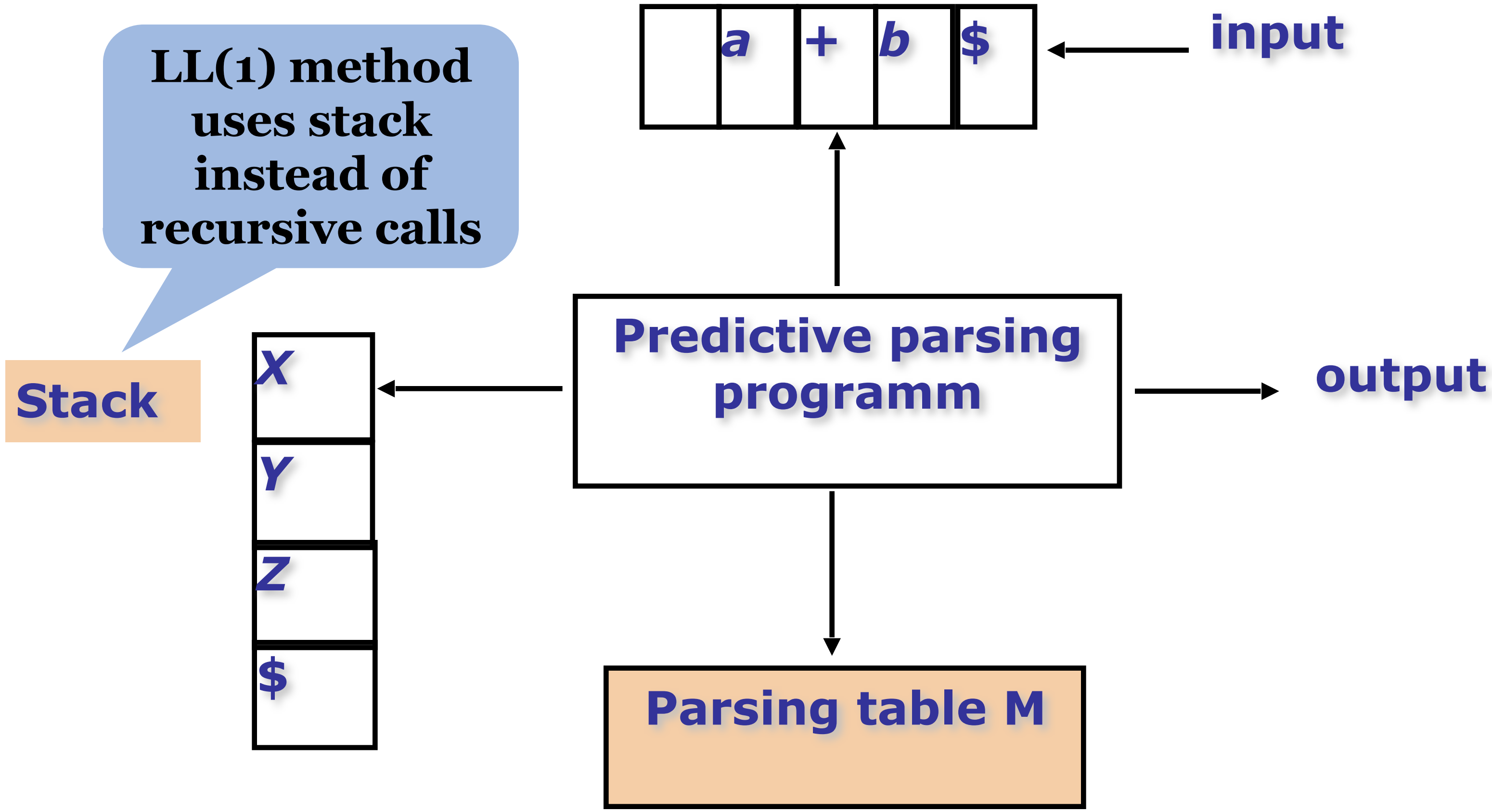
(4) It requires comput sets in order to detect the errors as early as p

Such as ")3-2)", the n 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

Main idea


LL(1) method uses stack instead of recursive calls
a
+
b
$
input
Stack
X
Y
Z
$
Predictive parsing programm
output
Parsing table M

Main idea

LL(1) Parsing **uses an explicit stack** rather than recursive calls to perform a parse, the parser can be visualized quickly and easily.

For example:

a simple grammar for the strings of balanced parentheses:

$$S \rightarrow (S)\ S \mid \varepsilon$$

- 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

Actions can be decided by a Parsing table which will be introduced later

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

Two Actions

- **The two actions**
  - Generate: Replace a non-terminal A at the top of the stack by a string α(in reverse) using a grammar rule $A \rightarrow \alpha$, and
  - Match: Match a token on top of the stack with the next input token.

- The list of generating actions in the above table:

$$
\begin{aligned}
S &\Rightarrow (S)S \quad [S \rightarrow (S)\ S] \\
&\Rightarrow (\ )S \quad [S \rightarrow \varepsilon] \\
&\Rightarrow (\ ) \quad [S \rightarrow \varepsilon]
\end{aligned}
$$

- 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

Purpose and Example of LL(1) 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

The General Definition of Table

- 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 represent potential errors

Table-Constructing Rule

- The table-constructing rule
  - ➢ If $A \rightarrow \alpha$ is a production choice, and there is a derivation $\alpha =>^{*} a\beta$, where a is a token, then add $A \rightarrow \alpha$ to the table entry $M[A,a]$;
  - ➢ If $A \rightarrow \alpha$ is a production choice, and there are derivations $\alpha =>^{*} \varepsilon$ and $S\$ =>^{*} \beta A a \gamma$, where S is the start symbol and a is a token (or $\$$), then add $A \rightarrow \alpha$ to the table entry $M[A,a]$;

A Table-Constructing Case

- **The constructing-process of the following table**
  - ➢ For the production : $S \to (S)\ S$, $\alpha=(S)S$, where $a=($, this choice will be added to the entry $M[S, (]$;
  - ➢ Since: $S \Rightarrow (S)S\varepsilon$, rule 2 applied with $\alpha= \varepsilon$, $\beta=($, $A = S$, $a =)$, and $\gamma=S\$$, so add the choice $S\to\varepsilon$ to $M[\ S,)\ ]$
  - ➢ Since $S\$ \Rightarrow^* S\$$, $S\to\varepsilon$ is also added to $M[S, \$]$.

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

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 one production in each table entry

- An LL(1) grammar cannot be ambiguous

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;

A Parsing Algorithm Using the LL(1) Parsing Table

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.

Example: If-Statements

- **The LL(1) parsing table for simplified grammar of if-statements:**

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

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

- 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

over

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

The parsing based LL(1) Table

- The parsing actions for the string:
  If (0) if (1) other else other

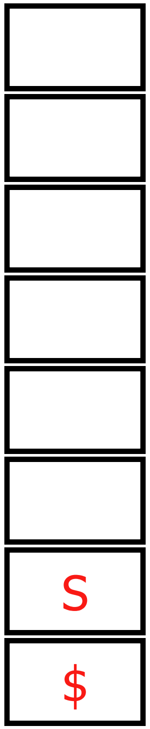
- (for conciseness, statement= S, if-stmt=I, else-part=L, exp=E, if=I, else=e, other=o)

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

S → I | o
I → i (E) S L
L→ e S | ε
E → 0 | 1

If (0) if (1) other else other

S

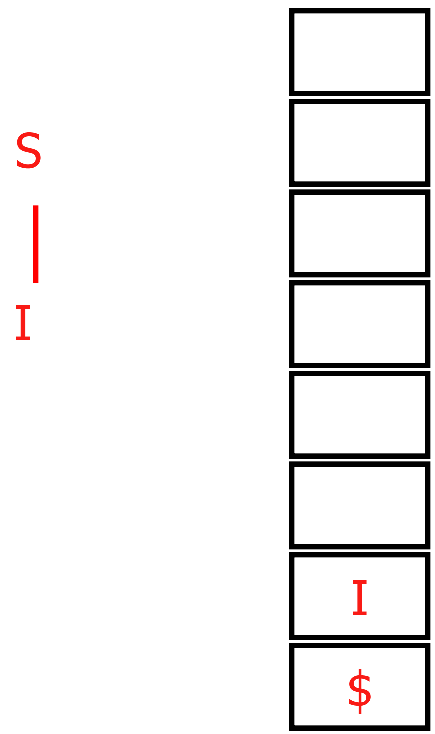


Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

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

S → I | o
I → i (E) S L
L→ e S | ε
E → 0 | 1

If (0) if (1) other else other

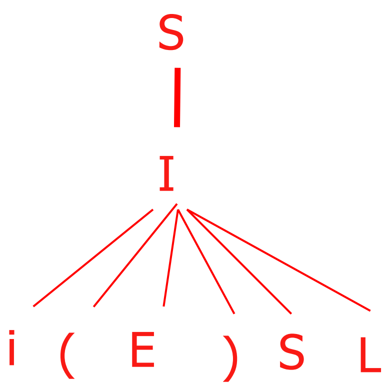


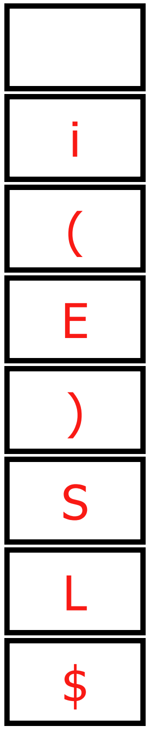
Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

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

S → I | o
I → i (E) S L
L→ e S | ε
E → 0 | 1

If (0) if (1) other else other



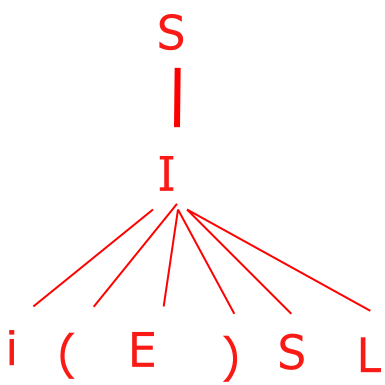


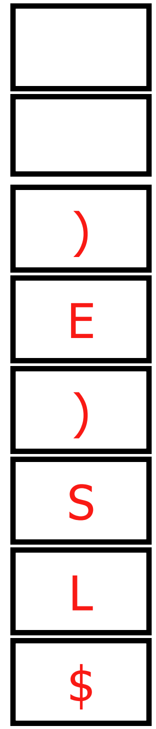
Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

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

S → I | o
I → i (E) S L
L→ e S | ε
E → 0 | 1

If (0) if (1) other else other



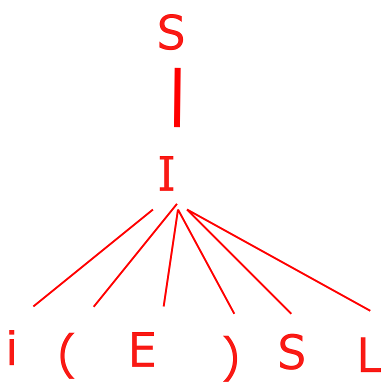


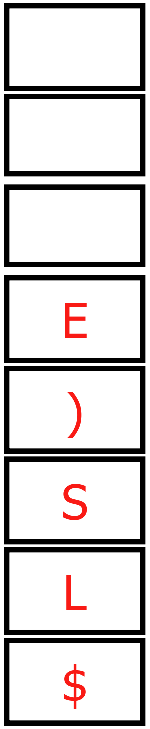
Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

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

S → I | o
I → i (E) S L
L→ e S | ε
E → 0 | 1

If (0) if (1) other else other



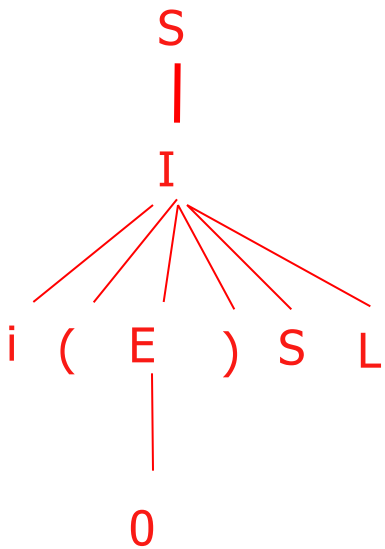


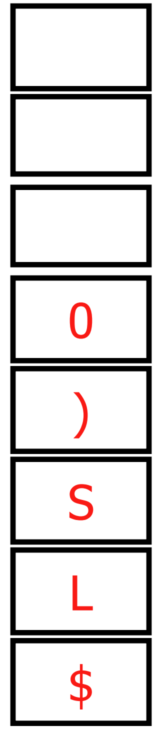
Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

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

S → I | o
I → i (E) S L
L→ e S | ε
E → 0 | 1

If (0) if (1) other else other



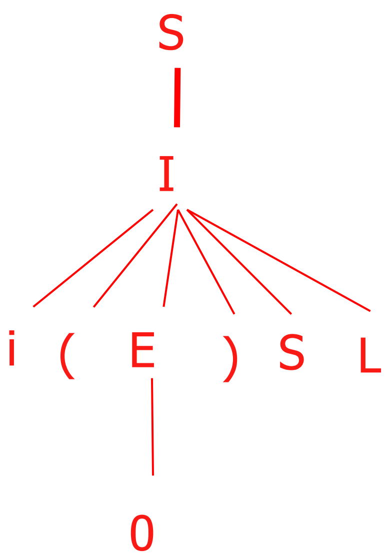


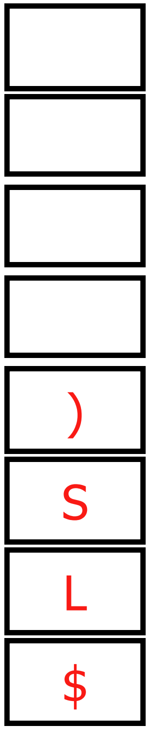
Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

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

$S \rightarrow I \mid o$
$I \rightarrow i\ (E)\ S\ L$
$L \rightarrow e\ S \mid \varepsilon$
$E \rightarrow 0 \mid 1$

If (0) if (1) other else other



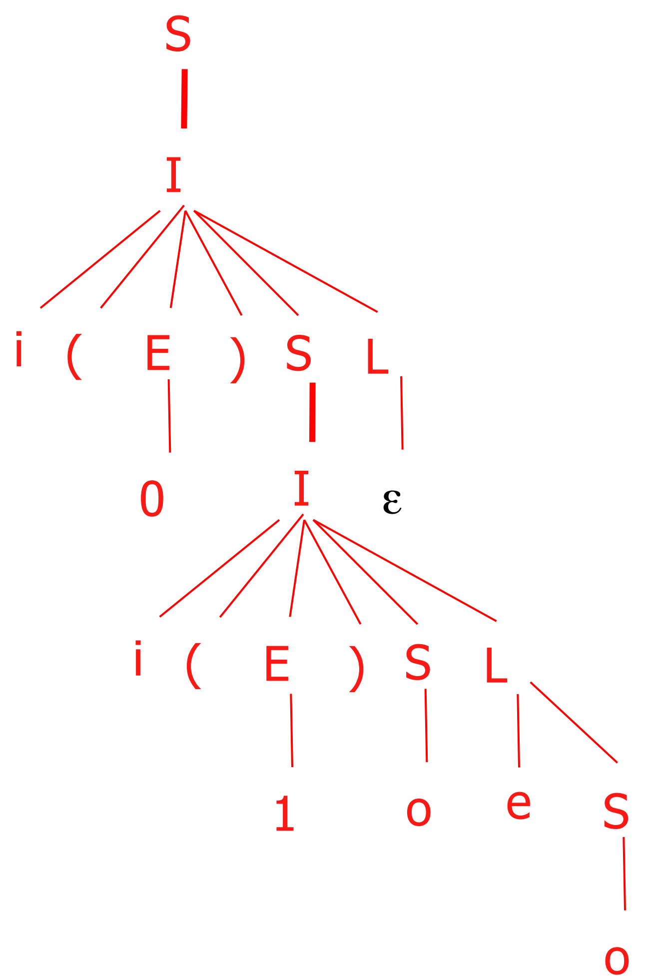


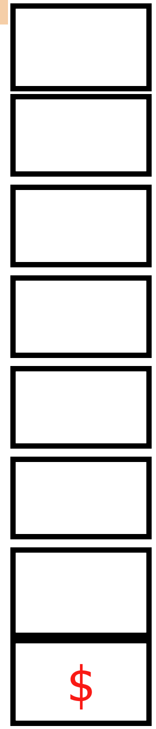
Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo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

The last Step:

We omit the procedure, and the last status of the stack and the parse tree is as follows:

If (0) if (1) other else other





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
	$ LS)0	0)i(1)oeo $	Match
	$ LS)	)i(1)oeo $	Match
	$ LS	i(1)oeo $	S→I
	$ LI	i(1)oeo $	I→i(E)SL
	$ LLS)E(i	i(1)oeo $	Match
	$ LLS)E(	(1)oeo	Match
	…	…	E→1
			Match
			match
			S→o
			match
			L→eS
			Match
			S→o
			match
			L→ε
22	$	$	accept

4.2.3 Left Recursion Removal and Left Factoring

Repetition and Choice Problem

- **Repetition and choice in LL(1) parsing suffer from similar problems** to be those that occur in recursive-descent parsing:
  The grammar is ambiguous and less of deterministic.

- **Solutions**：

1. Apply the same ideas of **using EBNF (in recursive-descent parsing)** to LL(1) parsing;
2. Rewrite the grammar within the BNF notation into a form that the LL(1) parsing algorithm can accept.

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])

Left Recursion Removal

- Left recursion is commonly used to make operations left associative

  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 |…

CASE 1: Simple Immediate Left Recursion

- $A \rightarrow A\alpha \mid \beta$
  Where, α and β are strings of terminals and non-terminals; β does not begin with A.

- The grammar will generate the strings of the form.

$$\beta\alpha^{n}$$

- We rewrite this grammar rule into two rules:

  $A \rightarrow \beta A'$

  To generate β first;

  $A' \rightarrow \alpha A' \mid \varepsilon$

  To generate the repetitions of α, using right recursion.

Example

- exp → exp addop term | term

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

  exp → term exp'

  exp' → addop term exp' | ε

CASE2: General Immediate Left Recursion

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

Where none of $\beta_1, \ldots, \beta_m$ begin with A.

The solution is similar to the simple case:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$

Example

- **exp → exp + term | exp - term |term**
- **Remove the left recursion as follows:**

  exp → term exp'

  exp' → + term exp' | - term exp' |ε

CASE3: General Left Recursion

- **Grammars with no ε-productions and no cycles**

  (1) A cycle is a derivation of at least one step that begins and ends with same non-terminal:

  $$A \Rightarrow \alpha \Rightarrow A$$

  (2) Programming language grammars do have ε-productions, but usually in very restricted forms.

Algorithm for General Left Recursion Removal

For i:=1 to m do
  For j:=1 to i-1 do
    Replace each grammar rule choice of the form
    $Ai \rightarrow Aj\beta$ by the rule
    $Ai \rightarrow \alpha_1\beta | \alpha_2\beta | \ldots | \alpha_k\beta$,
    where $Aj \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$ is the current rule for Aj.

Explanation:
(1) Picking an arbitrary order for all non-terminals, say, $A_1, \ldots, Am$;
(2) Eliminates all rules of the form $Ai \rightarrow Aj\gamma$ with $j \leq i$;
(3) Every step in such a loop would only increase the index, and thus the original index cannot be reached again.

Example

Consider the following grammar:

$A \rightarrow Ba \mid Aa \mid c$

$B \rightarrow Bb \mid Ab \mid d$

Where, $A_1 = A$, $A_2 = B$ and $m = 2$

(1) When $i=1$, the inner loop does not execute, So only to remove the immediate left recursion of A

$A \rightarrow BaA' \mid c\,A'$

$A' \rightarrow aA' \mid \varepsilon$

$B \rightarrow Bb \mid Ab \mid d$

Example

(2) when $i=2$, the inner loop execute once, with $j=1$;To eliminate the rule $B \to Ab$ by replacing A with it choices

$A \to BaA' \mid c\,A'$

$A' \to aA' \mid \varepsilon$

$B \to Bb \mid BaA'b \mid cAb \mid d$

(3) We remove the immediate left recursion of B to obtain

$A \to BaA' \mid c\,A'$

$A' \to aA' \mid \varepsilon$

$B \to |cA'bB' \mid dB'$

$B \to bB' \mid aA'bB' \mid \varepsilon$

Now, the grammar has no left recursion.

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

expr → expr addop term|term

addop → +|-

term → term mulop factor | factor

mulop →*

factor →(expr) | number

After removal of the left recursion

exp → term exp'

exp'→ addop term exp'|ε

addop → + -

term → factor term'

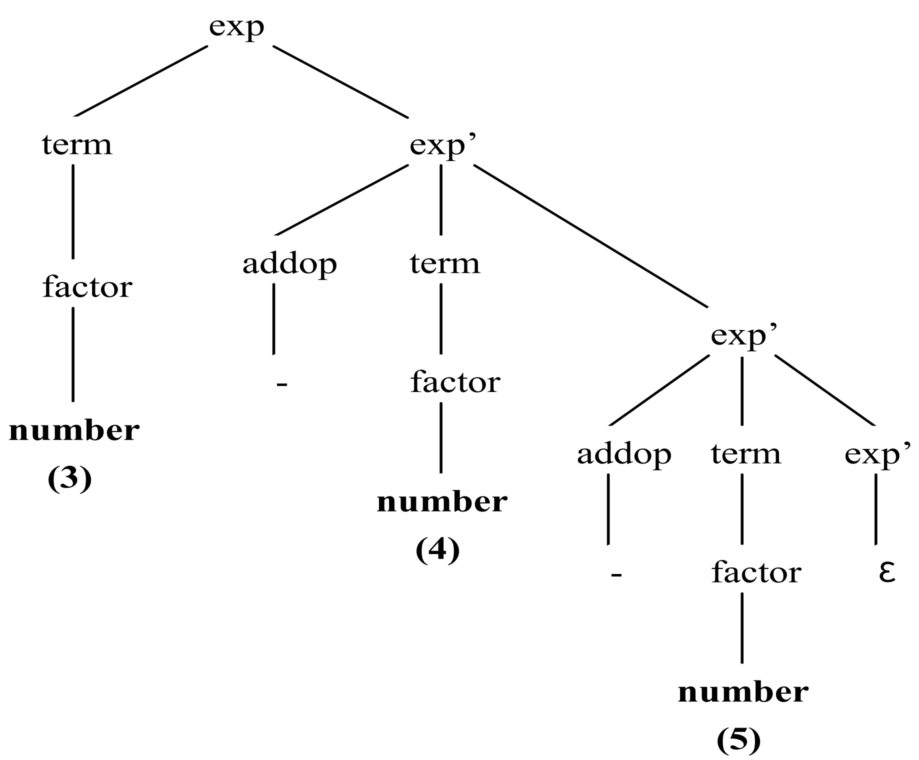
term' → mulop factor term'|ε

mulop →*

factor →(expr) | number

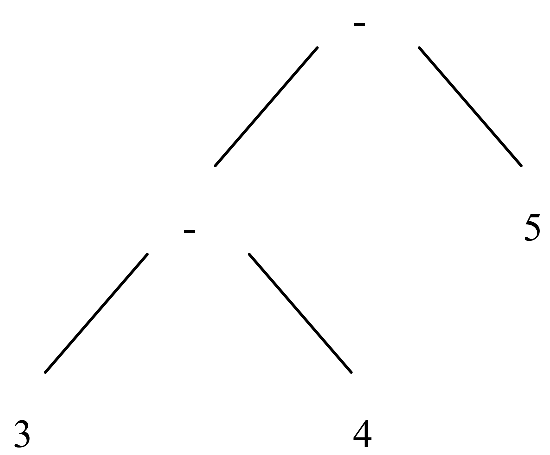
Parsing Tree

- The parse tree for the expression 3-4-5
  Not express the left associativity of subtraction.



Syntax Tree

- Nevertheless, a parse should still construct the appropriate left associative syntax tree



- From the given parse tree, we can see how the value of 3-4-5 is computed.

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'
```

Left-Recursion Removed Grammar and its Procedures

- To compute the value of the expression, exp' needs a parameter from the exp procedure

  exp → term exp'

  exp'→ addop term exp'|ε

```
function exp:integer;
    var temp:integer;
    Begin
     Temp:=Term;
     Return Exp'(temp);
    End exp;
```

```
function exp'(valsofar:integer):integer;
   Begin
    If token=+ or token=- then
    Case token of
     +: match(+);
       valsofar:=valsofar+term;
     -: match(-);
       valsofar:=valsofar-term;
    end case;
    return exp'(valsofar);
```

The LL(1) parsing table for the new expression

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					

Left Factoring

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

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

Example:

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

- 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:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

Algorithm for Left Factoring a Grammar

While there are changes to the grammar do

For each non-terminal ***A*** do

Let $\boldsymbol{\alpha}$ be a prefix of maximal length that is shared by two or more production choices for A

If $\boldsymbol{\alpha \neq \varepsilon}$ then

Let $\boldsymbol{A \rightarrow \alpha 1 | \alpha 2 | \ldots | \alpha n}$ be all the production choices for ***A***

And suppose that $\boldsymbol{\alpha 1, \alpha 2, \ldots, \alpha k}$ share $\boldsymbol{\alpha}$, so that

$$\boldsymbol{A \rightarrow \alpha\beta 1 | \alpha\beta 2 | \ldots | \alpha\beta k | \alpha K{+}1 | \ldots | \alpha n},$$

the $\boldsymbol{\beta j}$**'s** share No common prefix, and $\boldsymbol{\alpha K{+}1, \ldots, \alpha n}$ do not share $\boldsymbol{\alpha}$

Replace the rule $\boldsymbol{A \rightarrow \alpha 1 | \alpha 2 | \ldots | \alpha n}$ by the rules

$$\boldsymbol{A \rightarrow \alpha A' | \alpha K{+}1 | \ldots | \alpha n}$$

$$\boldsymbol{A' \rightarrow \beta 1 | \beta 2 | \ldots | \beta k}$$

Example 4.4

- Consider the grammar for statement sequences, written in right recursive form:

  Stmt-sequence→stmt; stmt-sequence | stmt

  Stmt→s

- Left Factored as follows:

  Stmt-sequence→stmt stmt-seq'

  Stmt-seq'→; stmt-sequence | ε

Example 4.4

- **Notices:**

  If we had written the stmt-sequence rule left recursively:

  Stmt-sequence→stmt-sequence ;stmt | stmt

  Then removing the immediate left recursion would result in the same rules:

  Stmt-sequence→stmt stmt-seq'

  Stmt-seq'→; stmt-sequence | ε

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

Example 4.6

- An arithmetic expression grammar with right associativity operation:

$$exp \rightarrow term+exp \mid term$$

- This grammar needs to be left factored, and we obtain the rules

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow +\ exp \mid \varepsilon$$

- Suppose we substitute term exp' for exp, we then obtain:

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow +\ term\ exp' \mid \varepsilon$$

Example 4.7

- An typical case where a grammar fails to be LL(1)

Statement → assign-stmt| call-stmt| other
Assign-stmt→identifier:=exp
Call-stmt→indentifier(exp-list)

Where, identifier is shared as first token of both ***assign-stmt*** and ***call-stmt*** and, thus, could be the lookahead token for either. But not in the form can be left factored.

Example 4.7

- First replace ***assign-stmt*** and ***call-stmt*** by the right-hand sides of their definition productions:

  Statement → identifier :=

  exp | indentifier(exp-list)| other

- Then, we left factor to obtain

  Statement → identifier statement' | other

  Statement' →:=exp |(exp-list)

- **Note:**

  This **obscures the semantics** of call and assignment by separating the identifier from the actual call or assign action.

4.2.4 Syntax Tree Construction in LL(1) Parsing

Difficulty in Construction

- It is more difficult for LL(1) to adapt to syntax tree construction than recursive descent parsing
- The structure of the syntax tree can be obscured by left factoring and left recursion removal
- The parsing stack represents only predicated structure, not structure that have been actually seen

Solution

- **The solution**

Delay the construction of syntax tree nodes to the point when structures are removed from the parsing stack.

An extra stack is used to keep track of syntax tree nodes, and the “action” markers are placed in the parsing stack to indicate when and what actions on the tree stack should occur

Example

- A barebones expression grammar with only an addition operation.

$$E \rightarrow E + n \mid n$$

/* be applied left association*/

- The corresponding LL(1) grammar with left recursion removal is:

$$E \rightarrow n\ E'$$
$$E' \rightarrow +nE' \mid \varepsilon$$

To compute the arithmetic value of the expression

- Use a separate stack to store the intermediate values of the computation, called the value stack; Schedule two operations on that stack:
  - ➢A push of a number;
  - ➢The addition of two numbers.

PUSH can be performed by the match procedure, and ADDITION should be scheduled on the stack, by pushing a special symbol (such as #) on the parsing stack.

This symbol must also be added to the grammar rule that match a +, namely, the rule for E': $\boldsymbol{E' \to +n\#E' | \varepsilon}$

- Notes: The addition is scheduled just after the next number, but before any more E' non-terminals are processed. This guaranteed left associativity.

The actions of the parser to compute the value of the expression 3+4+5

Parsing Stack	Input	Action	Value Stack
$E	3+4+5$	E→n E’	$
$E’n	3+4+5$	Match/push	$
$E’	+4+5$	E’ →+n#E’	3$
$E’#n+	+4+5$	Match	3$
$E’#n	4+5$	Match/push	3$
$E’#	+5$	Addstack	43$
$E’	+5$	E’ →+n#E’	7$
$E’#n+	+5$	Match	7$
$E’#n	5$	Match/push	7$
$E’#	$	Addstack	57$
$E’	$	E’ → ε	12$
$	$	Accept	12$

End of Part One

THANKS