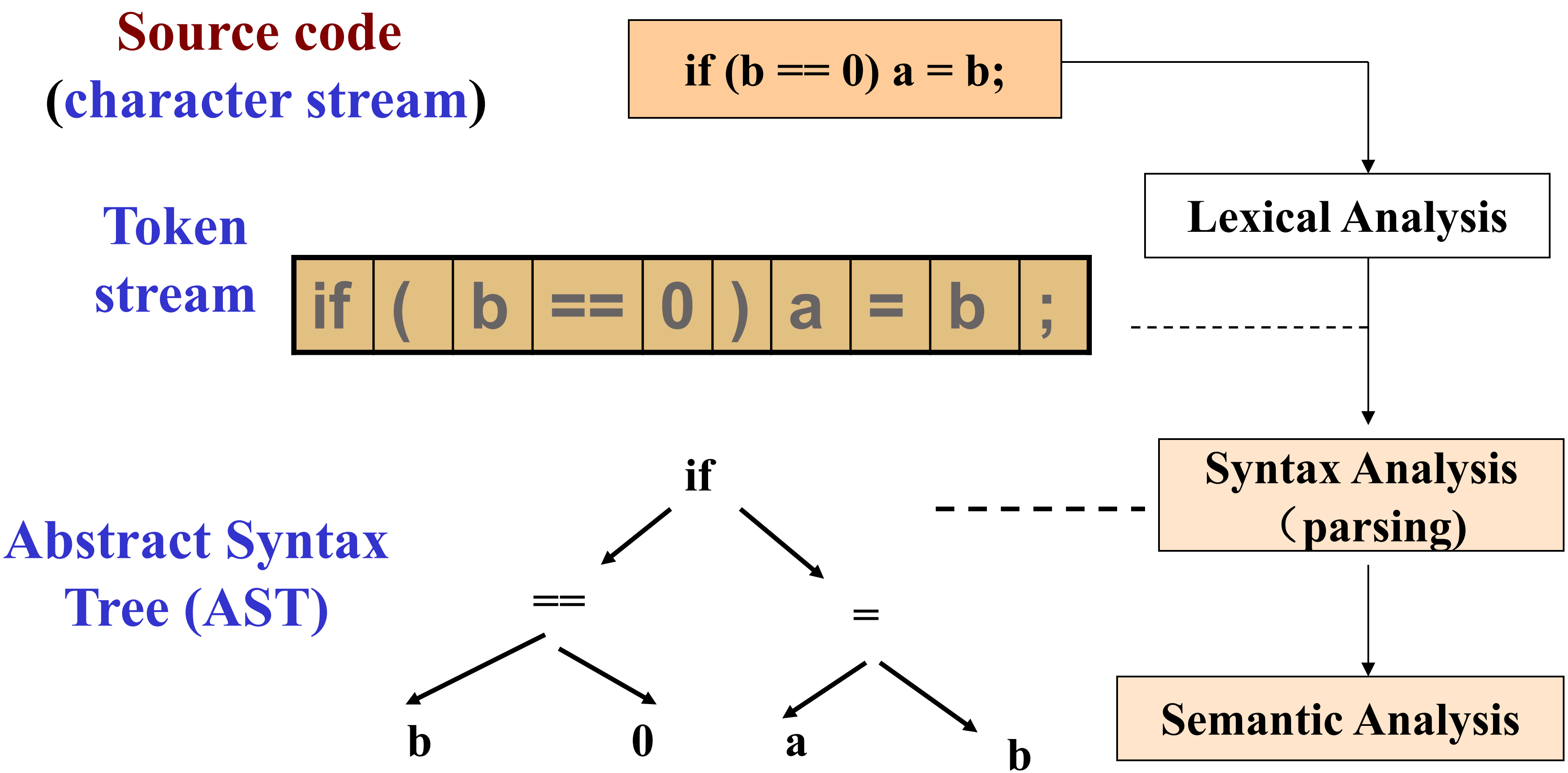
Chapter 4
Top-Down Parsing

2022 Spring&Summer

Outline

- Top-down parsing
- LL(k) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing

Where We Are



Source code
(character stream)
if (b == 0) a = b;
Lexical Analysis
Token
stream
if
(
b
==
0
)
a
=
b
;
Syntax Analysis
（parsing)
Abstract Syntax
Tree (AST)
if
==
=
b
0
a
b
Semantic Analysis

4.1 Top-Down Parsing by Recursive -Descent

- The idea of Recursive-Descent Parsing
  - 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.

4.1.1 The Basic Method of Recursive-Descent

- The first example: the *expression* Grammar:

  expr → *expr addop term*|*term*

  addop → + -

  term → *term mulop factor* | *factor*

  mulop → *

  factor → (*expr*) | *number*

- A recursive-descent procedure that recognizes a *factor* is as follows (in pseudo-code):

4.1.1 The Basic Method of Recursive-Descent

factor →***(****expr****)*** **| number**

```
Procedure factor
BEGIN
  case token of
  ( :   match( ( );
          expr;
          match( ));
  number:
        match(number);
  else error;
  end case;
END factor
```

- Writing recursive-decent procedure for the remaining rules in the expression grammar is not as easy for *factor*.
- It requires the use of EBNF.

4.1.2 Repetition and Choice: Using EBNF

- Example

  If-stmt → if (exp) statement

  | if (exp) statement else statement

- The EBNF of the if-statement is as follows:

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

4.1.2 Repetition and Choice: Using EBNF

- Example: $expr \to expr\ addop\ term \mid term$

  this would lead to an immediate infinite recursive loop.

  both *exp* and *term* can begin with the same tokens (a number or left parenthesis).

- The solution is to use the EBNF rule:

  $expr \to term\ \{addop\ term\}$

4.1.2 Repetition and Choice: Using EBNF

expr → *term* {*addop term*}

```
Procedure exp;
Begin
   term;
   while token = + or token = - do
      match(token);
      term;
   End while;
 End exp;
```

4.1.2 Repetition and Choice: Using EBNF

- EBNF rule for term:

  $term \rightarrow factor \ \{mulop \ factor\}$

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

4.1.2 Repetition and Choice: Using EBNF

One question: whether the *left associativity* implied by the curly bracket (and explicit in the original BNF) can still be maintained within this code.

4.1.2 Repetition and Choice: Using EBNF

- A recursive-descent calculator for the simple integer arithmetic of our grammar:

```
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;
```

4.1.3 Further Decision Problems

The recursive-descent method is quite powerful and adequate to construct a complete parse. But we need 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. (First Sets.)

(3) It may be necessary to know what token legally coming from the non-terminal A, $A \rightarrow \varepsilon$. (Follow Sets.)

(4) It requires computing the First and Follow sets in order to detect the errors as early as possible.

Such as "*)3-2)*", (descend from *exp* to *term* to *factor*)

4.2 LL(1) PARSING

Main idea:

LL(1) Parsing uses an explicit stack rather than recursive calls to perform a parse.

Example: a simple grammar

$$S \rightarrow E + S \mid E$$

$$E \rightarrow \mathbf{num} \mid (S)$$

4.2.1 The Basic Method of LL(1) Parsing

$S \rightarrow E + S \mid E$

$E \rightarrow \textbf{num} \mid (S)$

Partly-derived String	Lookahead	parsed part unparsed part
S	(	(1+2+(3+4))+5
⇒ E+S	(	(1+2+(3+4))+5
⇒ (S) +S	1	(1+2+(3+4))+5
⇒ (E+S)+S	1	(1+2+(3+4))+5
⇒ (1+S)+S	2	(1+2+(3+4))+5
⇒ (1+E+S)+S	2	(1+2+(3+4))+5
⇒ (1+2+S)+S	2	(1+2+(3+4))+5
⇒ (1+2+E)+S	(	(1+2+(3+4))+5
⇒ (1+2+(S))+S	3	(1+2+(3+4))+5
⇒ (1+2+(E+S))+S	3	(1+2+(3+4))+5

4.2.1 The Basic Method of LL(1) Parsing

- Want to decide which production to apply based on next symbol
  - (1) $\mathbf{S} \Rightarrow \mathbf{E} \Rightarrow (\mathbf{S}) \Rightarrow (\mathbf{E}) \Rightarrow (1)$
  - (1)+2 $\mathbf{S} \Rightarrow \mathbf{E}\text{+S} \Rightarrow (\mathbf{S})\text{+S} \Rightarrow (\mathbf{E})\text{+S}$

    $\Rightarrow (1)\text{+}\mathbf{E} \Rightarrow (1)\text{+}2$
- Why is this hard?

4.2.1 The Basic Method of LL(1) Parsing

- *$S \to (S)\ S \mid \varepsilon$*

Step	Parsing	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

■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*,

(2) *Match*: match a token on top of the stack with the next input token.

4.2.1 The Basic Method of LL(1) Parsing

- The list of generating actions in the above table:

$$
\begin{array}{lll}
S & \Rightarrow (S)S & [S \rightarrow (S)\ S] \\
 & \Rightarrow (\ )S & [S \rightarrow \varepsilon] \\
 & \Rightarrow (\ ) & [S \rightarrow \varepsilon]
\end{array}
$$

- Which corresponds to the steps in a leftmost derivation of string *( )*.

This is the characteristic of top-down parsing.

- Constructing a parse tree:

Adding node construction actions as each non-terminal or terminal is push onto the stack.

4.2.2 The LL(1) Parsing Table and Algorithm

- Purpose of the LL(1) Parsing Table:

  To express the possible rule choices for a non-terminal *A* : *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

4.2.2 The LL(1) Parsing Table and Algorithm

- The general LL(1) Parsing table definition:

  The table is a two-dimensional array indexed by non-terminals and terminals

  The table contains production choices to use at the appropriate parsing step, which 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.

4.2.2 The LL(1) Parsing Table and Algorithm

- The table-constructing rule:

  (1) 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];

  (2) 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];

4.2.2 The LL(1) Parsing Table and Algorithm

- The constructing-process of the above table:

  (1) For the production : $S \rightarrow (S)\ S$, $\alpha = (S)S$, where a=(, the entry M[S,( ] ;

  (2) For the production: $S \rightarrow \varepsilon$, $\alpha = \varepsilon$ i.e. there are derivation $\alpha => \varepsilon$ and $S\$ =>^* \beta A a \gamma = (S)S\$$. where a=) or a=$. *So add the choice S→ε to the both M[S,)] and M[S,$].*

4.2.2 The LL(1) Parsing Table and Algorithm

- 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.

4.2.2 The LL(1) Parsing Table and Algorithm

- A Parsing Algorithm Using the LL(1) Parsing Table:

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 $A \rightarrow X_1 X_2 \ldots X_n$

4.2.2 The LL(1) Parsing Table and Algorithm

- A Parsing Algorithm Using the LL(1) Parsing Table:

```
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.
```

4.2.2 The LL(1) Parsing Table and Algorithm

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	

4.2.2 The LL(1) Parsing Table and Algorithm

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.

the production: *else-part → else statement* preferred over the production *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 .

4.2.2 The LL(1) Parsing Table and Algorithm

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

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

Step	Parsing	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
6	$ LS)**0**	**0)i(1)oeo$**	match
7	$ LS)	**)i(1)oeo$**	match
8	$ LS	**i(1)oeo$**	S→I
9	$ LI	**i(1)oeo$**	I→i（E）SL
10	$LLS)E(**i**	**i(1)oeo$**	match
11	$ LLS)E(	**(1)oeo$**	match
12	$ LLS)E	**1)oeo$**	E→1
13	$ LLS)**1**	**1)oeo$**	match

4.2.2 The LL(1) Parsing Table and Algorithm

14	$ LLS)	**)oeo**$	match
15	$ LLS	**oeo**$	S→o
16	$ LL**o**	**oeo**$	match
17	$ LL	**eo**$	L→eS
18	$ LS**e**	**eo**$	match
19	$ LS	**o**$	S→o
20	$ L**o**	**o**$	match
21	$ L	$	L→ε
22	$	$	accept

4.2.3 left Recursion Removal and Left Factoring

Two standard techniques for Repetition and Choice problems:

1、left Recursion removal

Immediate left recursion: the left recursion occurs only within the production of a single non-terminal.

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

Indirect left recursion:

$$A \rightarrow Bb \mid \ldots$$
$$B \rightarrow Aa \mid \ldots$$

4.2.3 left Recursion Removal and Left Factoring

- Case 1: Simple immediate left recursion

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

  Rewrite this grammar rule into two rules:

  $A \rightarrow \beta A'$

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

- Example:

  exp → *exp addop term* | *term*

  exp → *term exp'*

  exp' → *addop term exp'* | ε

4.2.3 left Recursion Removal and Left Factoring

- Case 2: 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:

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

  remove the left recursion as follows:

  $exp \rightarrow term\ exp'$

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

4.2.3 left Recursion Removal and Left Factoring

- Case 3: General left recursion

  Grammars with no ε-productions and no cycles.

  (1) A cycle is a derivation of at least one step the 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.

4.2.3 left Recursion Removal and Left Factoring

- 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

     $A_i \rightarrow A_j\beta$ by the rule

     $$A_i \rightarrow \alpha_1\beta | \alpha_2\beta | \ \dots\ | \alpha_k\beta,$$

     where $A_j \rightarrow \alpha_1 | \alpha_2 | \ \dots\ | \alpha_k$ is the current rule for A_j.

     remove, if necessary, immediate left recursion

  involving A_i

4.2.3 left Recursion Removal and Left Factoring

Example: consider the following grammar,

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

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

Where, $A1=A$, $A2=B$ and $n=2$

(1) When i=1, the inner loop does not execute,

So only to remove the immediate left recursion of A

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

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

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

4.2.3 left Recursion Removal and Left Factoring

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

$$A \rightarrow BaA' \mid c\,A'$$
$$A' \rightarrow aA' \mid \varepsilon$$
$$B \rightarrow Bb \mid BaA'b \mid cA'b \mid d$$

$$A \rightarrow BaA' \mid c\,A'$$
$$A' \rightarrow aA' \mid \varepsilon$$
$$B \rightarrow Bb \mid Ab \mid d$$

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

$$A \rightarrow BaA' \mid c\,A'$$
$$A' \rightarrow aA' \mid \varepsilon$$
$$B \rightarrow cA'bB' \mid dB'$$
$$B' \rightarrow bB' \mid aA'bB' \mid \varepsilon$$

- Now, the grammar has no left recursion.

4.2.3 left Recursion Removal and Left Factoring

- 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.

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

4.2.3 left Recursion Removal and Left Factoring

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

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

4.2.3 left Recursion Removal and Left Factoring

The grammar with its left recursion removed would give rise to the procedures exp and exp' as follows:

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

exp → term exp'

```
procedure exp
begin
  term;
  exp';
end exp;
```

exp'→ addop term exp'|*ε*

```
procedure exp'
begin
  case token of
   +: match(+);
     term;
     exp';
   -: match(-);
     term;
     exp';
  end case;
end exp'
```

4.2.3 left Recursion Removal and Left Factoring

The LL(1) parsing table for the new expression is given as follows:

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*→ (*exp*)	*factor*→ **number**					

4.2.3 left Recursion Removal and Left Factoring

2、 Left factoring:

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

$A \rightarrow \alpha\beta \mid \alpha\gamma \quad \Rightarrow \quad \boxed{\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta \mid \gamma \end{array}}$

4.2.3 left Recursion Removal and Left Factoring

- Example

  Stmt-sequence→stmt; stmt-sequence | stmt

  Stmt→s

Left Factored as follows:

Stmt-sequence→stmt stmt-seq'

Stmt-seq'→; stmt-sequence | ε

4.2.3 left Recursion Removal and Left Factoring

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 $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$ be all the production choices for A

And suppose that $\alpha_1, \alpha_2, \ldots, \alpha_k$ share α, so that

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_k | \alpha_{K+1} | \ldots | \alpha_n$, the β_j's share no common prefix, and $\alpha_{K+1}, \ldots \alpha_n$ do not share α

$A \rightarrow \alpha A' | \alpha_{K+1} | \ldots | \alpha_n$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_k$

4.2.3 left Recursion Removal and Left Factoring

- Example :

  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*

4.2.3 left Recursion Removal and Left Factoring

Example：

$$exp \rightarrow term{+}exp \mid term$$

$$exp \rightarrow term\ exp'$$

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

Suppose we substitute term exp' for exp:

$$exp \rightarrow term\ exp'$$

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

4.2.3 left Recursion Removal and Left Factoring

- Example

statement → *assign-stmt* | *call-stmt* | *other*
assign-stmt → *identifier:=exp*
call-stmt → *identifier(exp-list)*

statement → *identifier:=exp* | *identifier(exp-list)*
| *other*
statement → *identifier statement'* | *other*
statement' → *:=exp* | *(exp-list)*

Caution:
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

- It is more difficult for LL(1) to adapt to syntax tree construction.

  (1) The structure of the syntax tree can be obscured by left factoring and left recursion removal;

  (2) The parsing stack represents only predicated structure, not structure that have been actually seen.

The solution:

  (1) An extra stack is used to keep track of syntax tree nodes, and

  (2) "*action*" markers are placed in the parsing stack to indicate when and what actions on the tree stack should occur.

4.2.4 Syntax Tree Construction in LL(1) Parsing

- How to compute the arithmetic value of the expression.
  1. Use a separate stack to store the intermediate values of the computation, called the value stack;
  2. Schedule two operations on that stack:
     A push of a number;
     The addition of two numbers.
  3. PUSH can be performed by the match procedure, and
  4. ADDITION should be scheduled on the stack, by pushing a special symbol (such as #) on the parsing stack.

4.2.4 Syntax Tree Construction in LL(1) Parsing

- This symbol must also be added to the grammar rule that match a +,

  the rule for E':

  $E' \rightarrow +n\#E' | \varepsilon$

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

Homework of Chapter 4

4.8

Consider the grammar

lexp→atom|list

atom→number|***identifier***

list→(lexp-seq)

lexp-seq→lexp-seq lexp|lexp

(1) Remove the left recursion.

(2) Construct First and Follow sets for the nonterminals of the resulting gra mmar.

(3) Show that the resulting grammar is LL(1).

(4) Construct the LL(1) parsing table for the resulting grammar.

(5) Show the actions of the corresponding LL(1) parser,given the input stri ng(a (b (2)) (c)).

Homework of Chapter 4

4.12 Questions:

(1) Can an LL(1) grammar be ambiguous? Why or why not?

(2) Can an ambiguous grammar be LL(1)? Why or why not?

(3) Must an unambiguous grammar be LL(1)? Why or why not?