Chapter 5

Top-Down Parsing

Recursive Descent Parser

- Consider the grammar:

  $S \rightarrow c\ A\ d$

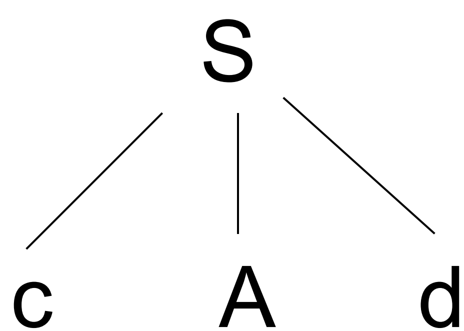
  $A \rightarrow ab \mid a$

  The input string is “*cad*”

Recursive Descent Parser (Cont.)

- Build parse tree:

  step 1. From start symbol.



Recursive Descent Parser (Cont.)

Step 2. We expand A using the first alternative $A \rightarrow ab$ to obtain the following tree:

```
        S
      / | \
     c  A  d
       / \
      a   b
```

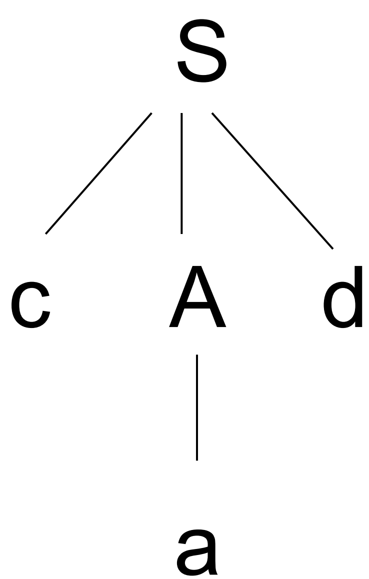
Recursive Descent Parser (Cont.)

- Now, we have a match for the second input symbol “a”, so we advance the input pointer to “d”, the third input symbol, and compare d against the next leaf “b”.

- Backtracking
  - Since “b” does not match “d”, we report failure and go back to A to see whether there is another alternative for A that has not been tried - that might produce a match!
  - In going back to A, we must reset the input pointer to “a”.

Recursive Descent Parser (Cont.)

Step 3.



Creating a top-down parser

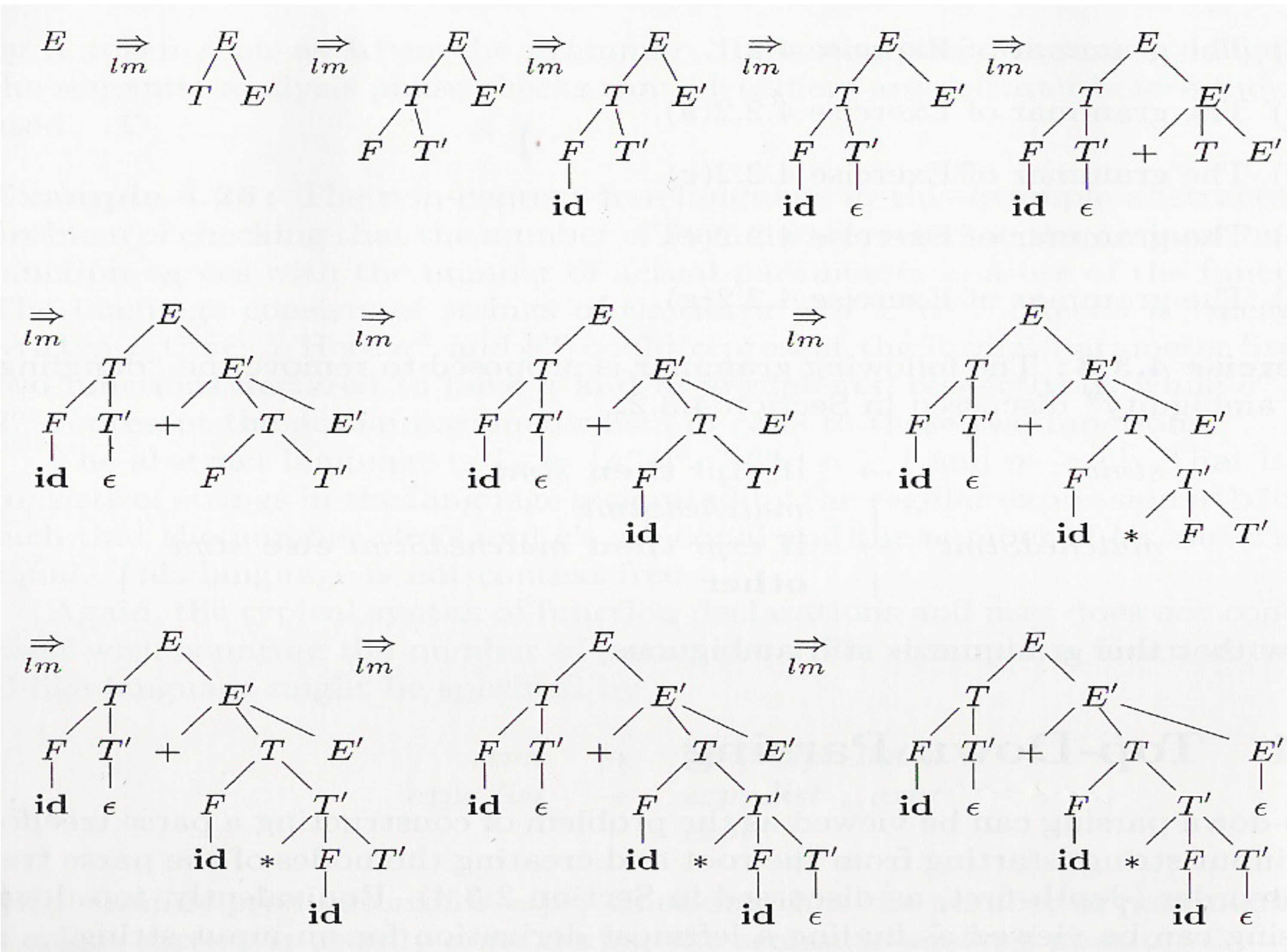
- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting form the root and creating the nodes of the parse tree in preorder.

- An example follows.

Creating a top-down parser (Cont.)

- Given the grammar :
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \lambda$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \lambda$
  - $F \rightarrow (E) \mid id$
- The input: id + id * id

Creating a top-down parser (Cont.)



Top-down parsing

- A top-down parsing program consists of a set of procedures, one for each non-terminal.
- Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

Top-down parsing

A typical procedure for non-terminal A in a top-down parser:

```
boolean A() {
   choose an A-production, A → X1 X2 … Xk;
   for (i= 1 to k) {
      if (Xi is a non-terminal)
           call procedure Xi();
      else if (Xi matches the current input token “a”)
           advance the input to the next token;
      else  /* an error has occurred */;
   }
}
```

NOTE

Top-down parsing

- Given a grammar:

input → expression
expression → term rest_expression
term → ID | parenthesized_expression
parenthesized_expression → ‘(‘ expression ‘)’
rest_expression → ‘+’ expression | λ

Top-down parsing

- For example:

  input:

$$ID + (ID + ID)$$

Top-down parsing

Build parse tree:

start from start symbol to invoke:

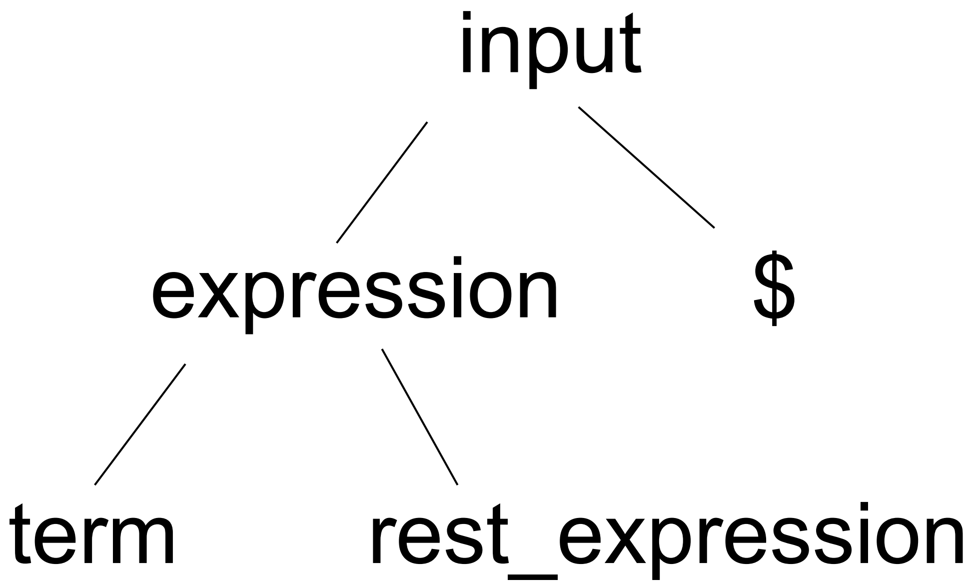
int input (void)

input

expression $

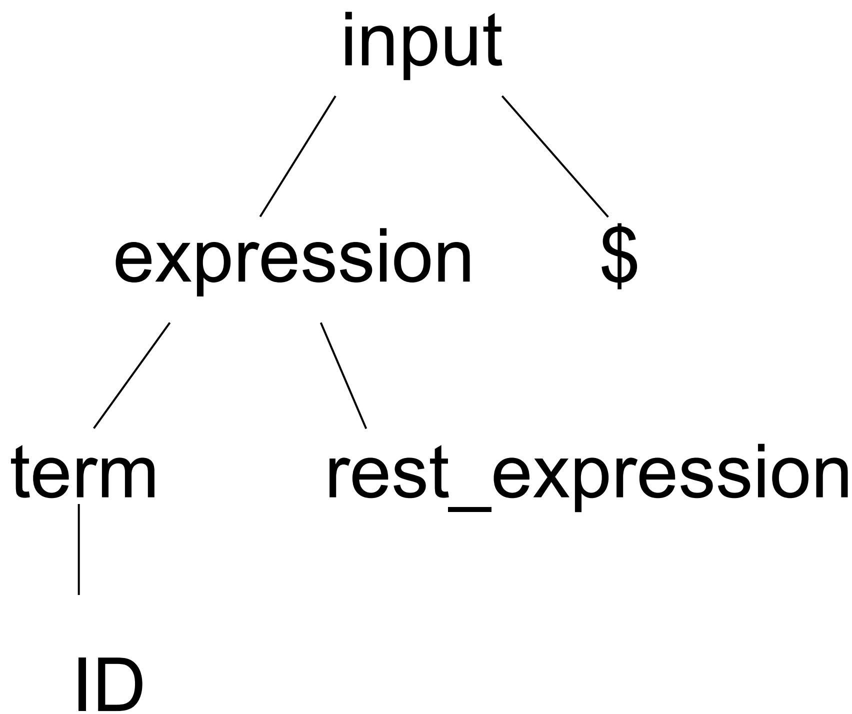
Next, invoke expression()

Top-down parsing



Next, invoke term()

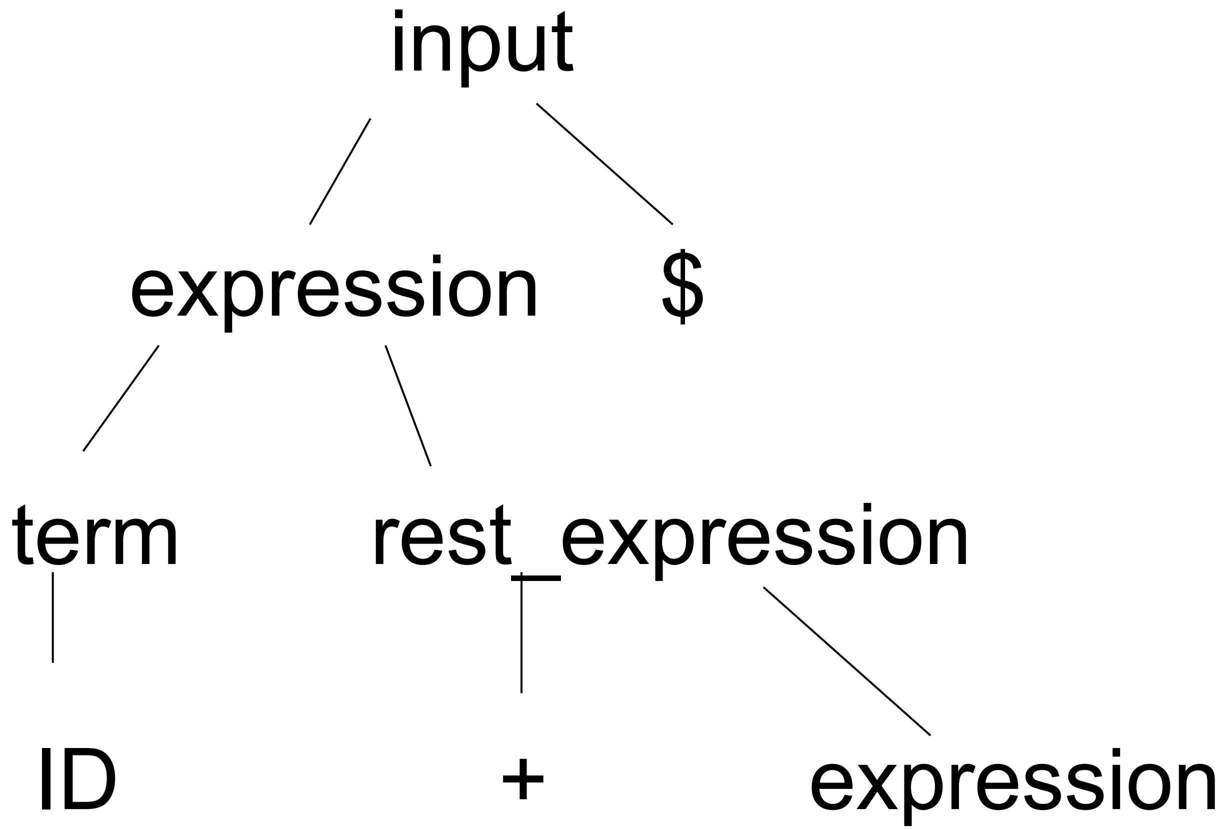
Top-down parsing



select term → ID (matching input string “ID”)

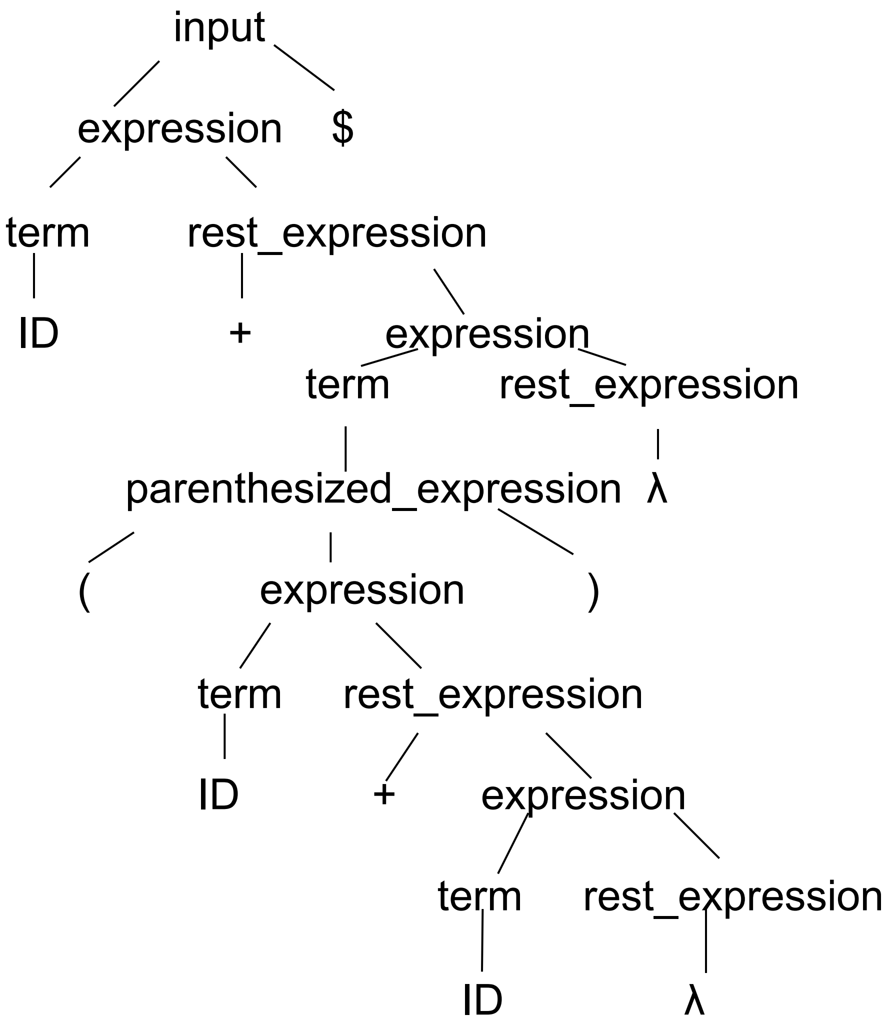
Top-down parsing

Invoke rest_expression()



Top-down parsing

The parse tree is:



LL(1) Parsers

- The class of grammars for which we can construct predictive parsers looking **k symbols ahead** in the input is called the LL(k) class.

- Predictive parsers, that is, recursive-descent parsers without backtracking, can be constructed for the LL(1) class grammars.

- The first "L" stands for scanning input from left to right. The second "L" for producing a leftmost derivation. The "1" for using one input symbol of look-ahead at each step to make parsing decisions.

LL(1) Parsers (Cont.)

$A \rightarrow \alpha \mid \beta$ are two distinct productions of grammar G, G is LL(1) if the following 3 conditions hold:

1. FIRST(α) cannot contain any terminal in FIRST(β).
2. At most one of α and β can derive λ.
3. if $\beta \rightarrow^* \lambda$, FIRST($\alpha$) cannot contain any terminal in FOLLOW(A).
   if $\alpha \rightarrow^* \lambda$, FIRST($\beta$) cannot contain any terminal in FOLLOW(A).

Some helping examples

- $S \to AC$

  $A \to a \mid B$ let's look at this production

  $B \to b \mid \lambda$

because A has two choices to derive, *a* and B, but B can $\to^* \lambda$ (that is λ is in First(B) = {b, λ}) so, if B does be derived into a λ, we can still have a chance if we can check the follow set of A. In other words, A must appear on RHS in other productions rule. Suppose A derive into a λ, we want to know what symbols follow A. In this example, FOLLOW(A) = FIRST(C);

Construction of a predictive parsing table

- The following rules are used to construct the predictive parsing table:
  - 1. for each terminal a in FIRST(α),

    add $A \rightarrow \alpha$ to matrix M[A,a]
  - 2. if λ is in FIRST(α), then

    for each terminal b in FOLLOW(A),

    add $A \rightarrow \alpha$ to matrix M[A,b]

LL(1) Parsers (Cont.)

Given the grammar:

input → expression		1
expression → term rest_expression		2
term → ID		3
	| parenthesized_expression	4
parenthesized_expression → ‘(‘ expression ‘)’		5
rest_expression → ‘+’ expression		6
	| λ	7

Build the parsing table.

LL(1) Parsers (Cont.)

FIRST (input) = FIRST(expression)

=FIRST (term) = {ID, ‘(‘ }

FIRST (parenthesized_expression) = { ‘(‘ }

FIRST (rest_expression) = { ‘+’ λ}

FOLLOW (input) = {$ }

FOLLOW (expression) = {$ ‘)’ }

FOLLOW (term) =

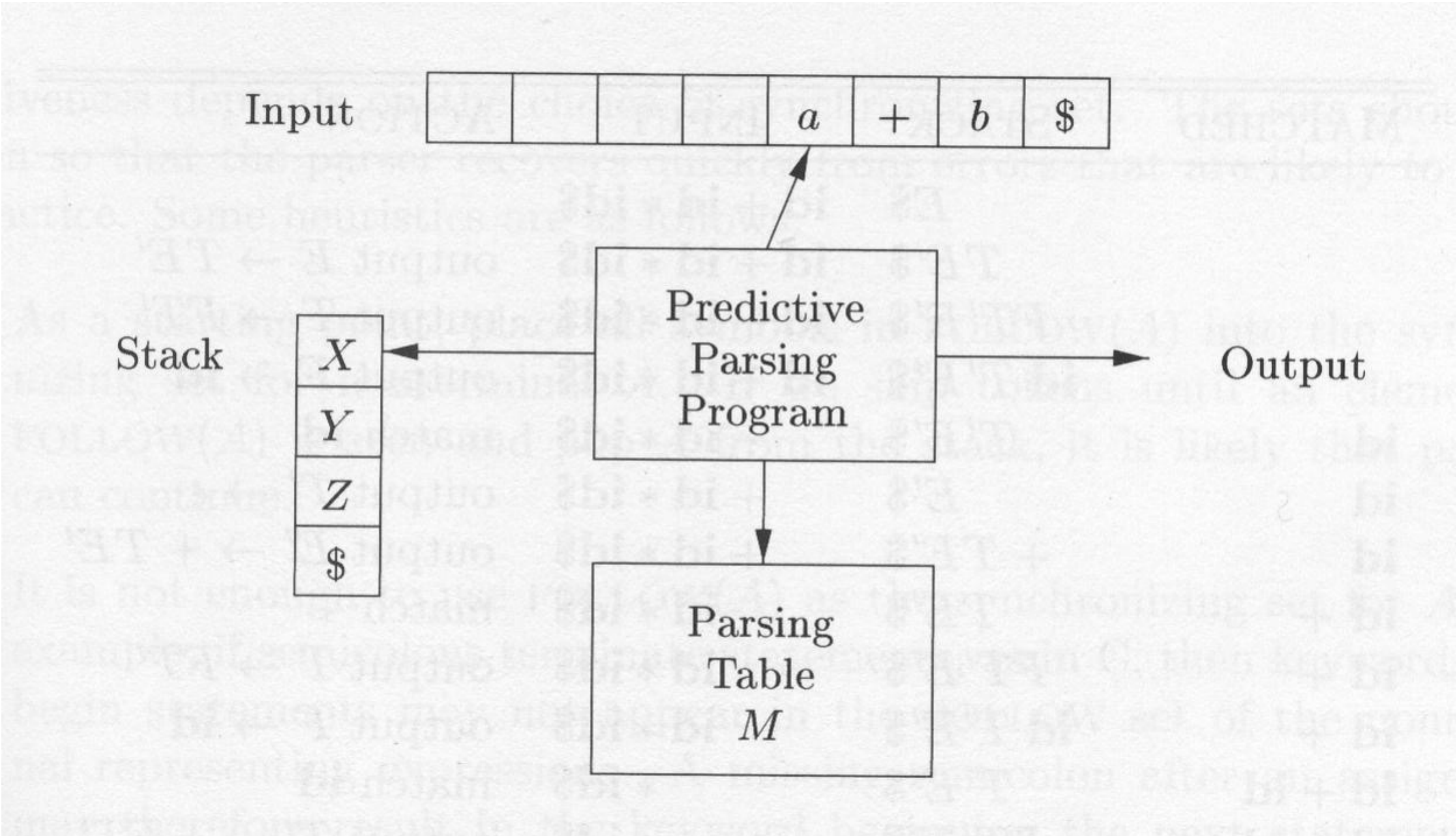
FOLLOW (parenthesized_expression) = {$ ‘+’ ‘)’}

FOLLOW (rest_expression) = {$ ‘)’}

LL(1) Parsers (Cont.)

Non-terminal	Input symbol				
	ID	+	(	)	$
Input	1		1		
Expression	2		2		
Term	3		4		
parenthesized_expression			5		
rest_expression		6		7	7

Model of a table-driven predictive parser



Predictive parsing algorithm

```
Set input pointer (ip) to the first token a;
Push $ and start symbol to the stack.
Set X to the top stack symbol;
while (X != $) { /*stack is not empty*/
  if (X is token a) pop the stack and advance ip;
  else if (X is another token) error();
  else if (M[X,a] is an error entry) error();
  else if (M[X,a] = X → Y1Y2…Yk) {
          output the production X → Y1Y2…Yk;
          pop the stack;          /* pop X */
         /* leftmost derivation*/
          push Yk,Yk-1,…, Y1 onto the stack, with Y1 on top;
  }
  set X to the top stack symbol Y1;
} // end while
```

LL(1) Parsers (Cont.)

- Given the grammar:
  - E → TE’ 1
  - E’ → +TE’ 2
  - E’ → λ 3
  - T → FT’ 4
  - T’ → *FT’ 5
  - T’ → λ 6
  - F → (E) 7
  - F → id 8

LL(1) Parsers (Cont.)

FIRST(F) = FIRST(T) = FIRST(E) = { (, id }

FIRST(E') = { +, λ }

FIRST(T') = { *, λ }

FOLLOW(E) = FOLLOW(E') = {), $ }

FOLLOW(T) = FOLLOW(T') = { +,), $ }

FOLLOW(F) = { +, *,), $ }

LL(1) Parsers (Cont.)

Non-terminal	Input symbols					
	Id	+	*	(	)	$
E	1			1		
E’		2			3	3
T	4			4		
T’		6	5		6	6
F	8			7		

LL(1) Parsers (Cont.)

Stack	Input	Output
$E	id + id * id $	
$E’T	id + id * id $	E → TE’
$E’T’F	id + id * id $	T → FT’
$E’T’id	id + id * id $	F → id
$E’T’	+ id * id $	match id
$E’	+ id * id $	T’ → λ
$E’T+	+ id * id $	E’ → +TE’

LL(1) Parsers (Cont.)

Stack	Input	Output
$E’T	id * id $	match +
$E’T’F	id * id $	T → FT’
$E’T’id	id * id $	F → id
$E’T’	* id $	match id
$E’T’F*	* id $	T’ → *FT’
$E’T’F	id $	match *
$E’T’id	id $	F → id
$E’T’	$	match id
$E’	$	T’ → λ
$	$	E’ → λ

Common Prefix

In Fig. 5.12, the common prefix:

if Expr then StmtList (R1,R2)

makes looking ahead to distinguish R1 from R2 hard.

Just use Fig. 5.13 to factor it and “var”(R5,6)

The resulting grammar is in Fig. 5.14.

1	Stmt	$\rightarrow$	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	$\rightarrow$	StmtList ; Stmt
4		|	Stmt
5	Expr	$\rightarrow$	var + Expr
6		|	var

Figure 5.12: A grammar with common prefixes.

---

procedure F ( )
    foreach $\mathsf{A} \in N$ **do**
        $\alpha \leftarrow LongestCommonPrefix(ProductionsFor(\mathsf{A}))$
        while $|\alpha| > 0$ **do**
            $V \leftarrow$ **new** *NonTerminal* ( )
            $Productions \leftarrow Productions \cup \{ \mathsf{A} \rightarrow \alpha V \}$
            foreach $p \in ProductionsFor(A) \mid RHS(p) = \alpha\beta_p$ **do** 13
                $Productions \leftarrow Productions - \{ p \}$
                $Productions \leftarrow Productions \cup \{ V \rightarrow \beta_p \}$
            $\alpha \leftarrow LongestCommonPrefix(ProductionsFor(\mathsf{A}))$
end

Figure 5.13: Factoring common prefixes.

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		|	else StmtList endif
4	StmtList	$\rightarrow$	StmtList ; Stmt
5		|	Stmt
6	Expr	$\rightarrow$	var V_2
7	V_2	$\rightarrow$	+ Expr
8		|	λ

Figure 5.14: Factored version of the grammar in Figure 5.12.

```
procedure E          L    R          ( )
    foreach A ∈ N do
        if ∃ r ∈ ProductionsFor(A) | RHS(r) = Aα
        then
            X ← new NonTerminal( )
            Y ← new NonTerminal( )
            foreach p ∈ ProductionsFor(A) do
                if p = r
                then Productions ← Productions ∪ { A → X Y }
                else Productions ← Productions ∪ { X → RHS(p) }
            Productions ← Productions ∪ { Y → αY, Y → λ }
end
```

Figure 5.15: Eliminating left recursion.

Left Recursion

- A production is **left recursive** if its LHS symbol is the first symbol of its RHS.

- In fig. 5.14, the production

  StmtList→ StmtList ; Stmt

  StmtList is left-recursion.

Left Recursion (Cont.)

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		|	else StmtList endif
4	StmtList	$\rightarrow$	StmtList ; Stmt
5		|	Stmt
6	Expr	$\rightarrow$	var V_2
7	V_2	$\rightarrow$	+ Expr
8		|	λ

Figure 5.14: Factored version of the grammar in Figure 5.12.

Left Recursion (Cont.)

- Grammars with left-recursive productions can never be LL(1).
  - Some look-ahead symbol t predicts the application of the left-recursive production

    $$A \rightarrow A\beta.$$

    with **recursive-descent parsing**, the application of this production will cause

    procedure A to be invoked infinitely.

  Thus, we must eliminate left-recursion.

Left Recursion (Cont.)

Consider the following left-recursive rules.

1. $A \rightarrow A\ \alpha$
2. $\quad |\ \beta$ the rules produce strings like $\beta\ \alpha\ \alpha$

we can change the grammar to:

1. $A \rightarrow X\ Y$
2. $X \rightarrow \beta$
3. $Y \rightarrow \alpha\ Y$
4. $\quad |\ \lambda$ the rules also produce strings like $\beta\ \alpha\ \alpha$

The EliminateLeftRecursion algorithm is shown in fig. 5.15. Applying it to the grammar in fig. 5.14 results in fig. 5.16.

Left Recursion (Cont.)

procedure ELIMINATELEFTRECURSION( )
  foreach $\mathsf{A} \in N$ **do**
    if $\exists\, r \in \mathit{ProductionsFor}(\mathsf{A}) \mid \mathit{RHS}(r) = \mathsf{A}\alpha$ 1
    then
      $X \leftarrow$ **new** *NonTerminal*( )
      $Y \leftarrow$ **new** *NonTerminal*( )
      foreach $p \in \mathit{ProductionsFor}(\mathsf{A})$ **do** 2
        if $p = r$
        then $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, \mathsf{A} \rightarrow X\ Y \,\}$ 3
        else $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, X \rightarrow \mathit{RHS}(p) \,\}$ 4
      $\mathit{Productions} \leftarrow \mathit{Productions} \cup \{\, Y \rightarrow \alpha Y,\ Y \rightarrow \lambda \,\}$ 5
end

Figure 5.15: Eliminating left recursion.

Left Recursion (Cont.)

Now, we trace the algorithm with the grammar below:

(4) StmtList → StmtList ; Stmt

(5) | Stmt

first, the input is (4) StmtList → StmtList ; Stmt

because RHS(4) = StmtList α it is left-recursive (marker 1)

create two non-terminals X, and Y

for rule (4) (marker 2)

as StmtList = StmtList,

create StmtList → XY (marker 3)

for rule (5) (marker 2)

as StmtList != Stmt

create X → Stmt (marker 4)

finally, create Y → ; Stmt and Y → λ (marker 5)

Left Recursion (Cont.)

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		|	else StmtList endif
4	StmtList	$\rightarrow$	X Y
5	X	$\rightarrow$	Stmt
6	Y	$\rightarrow$	; Stmt Y
7		|	λ
8	Expr	$\rightarrow$	var V_2
9	V_2	$\rightarrow$	+ Expr
10		|	λ

4	StmtList	$\rightarrow$	StmtList ; Stmt
5		|	Stmt

re 5.16: LL(1) version of the grammar in Figure 5.14.

Homework 1

Construct the LL(1) table for the following grammar:

1 $\text{Expr} \rightarrow \text{- Expr}$
2 $\text{Expr} \rightarrow \text{(Expr)}$
3 $\text{Expr} \rightarrow \text{Var ExprTail}$
4 $\text{ExprTail} \rightarrow \text{- Expr}$
5 $\text{ExprTail} \rightarrow \lambda$
6 $\text{Var} \rightarrow \text{id VarTail}$
7 $\text{VarTail} \rightarrow \text{(Expr)}$
8 $\text{VarTail} \rightarrow \lambda$

Homework 1 Solution

First(Expr) = {-, (, id}

First(ExprTail) = {-, λ}

First (Var) = { id}

First (VarTail) = { (, λ}

Follow (Expr) = Follow (ExprTail) = {$,) }

Follow (Var) = {$,), -}

Follow (VarTail) = {$,), -}

Homework 1 Solution (Cont.)

Non-Terminal	Input Symbol				
	-	(	id	)	$
Expr	1	2	3		
ExprTail	4			5	5
Var			6		
VarTail	8	7		8	8

Homework 2

- Given the grammar:
  - $S \rightarrow i\ E\ t\ S\ S' \mid a$
  - $S' \rightarrow e\ S \mid \lambda$
  - $E \rightarrow b$

  - 1. Find the first set and follow set.
  - 2. Build the parsing table.

Homework 2 Solution

First(S) = {i, a}

First(S') = {e, λ}

First (E) = {b}

Follow (S) = Follow (S') = {$, e}

Follow (E) = {t}

Homework 2 Solution (Cont.)

Non-Terminal	Input Symbol					
	a	b	e	i	t	$
S	2			1		
S'			3/4			4
E		5				

As First(S') contains λ and Follow (S') = {$, e} So rule 4 is added to e, $.

3/4 (rule 3 or 4) means an error. This is not LL(1) grammar.