Chapter 4 Syntax Analysis — Top-Down Parsing

Instructor: Zhen Gao
gaozhen@tongji.edu.cn

Outline

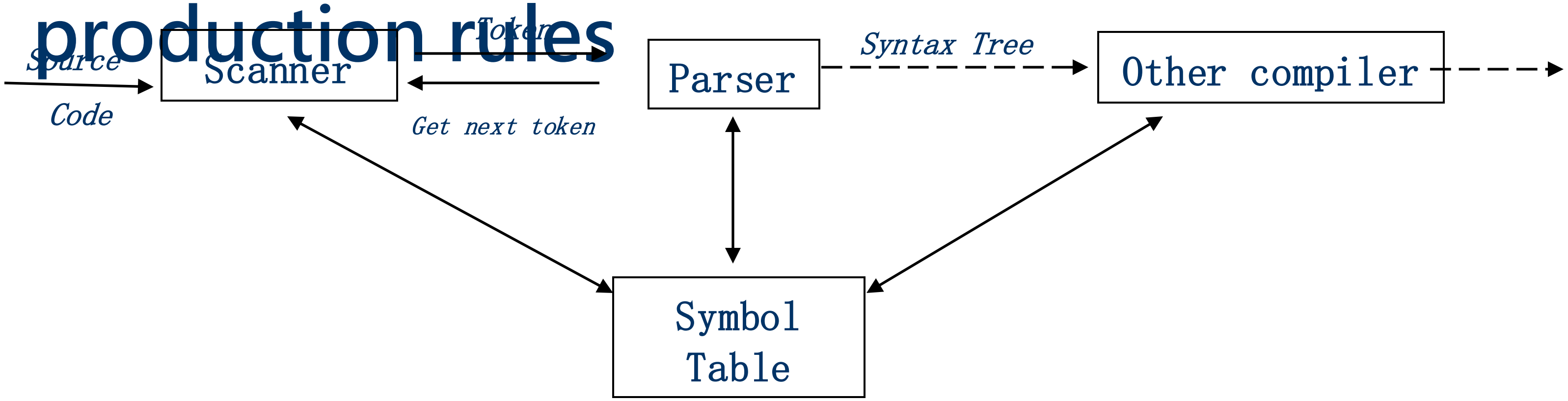
- Functions of a parser
- Overview of top-down parsing
- LL(1) parsing method
- Recursive descent parser
- Predictive parser

Parser

- Task：

  For any given w ∈ V_T*，determine if w ∈ L(G)?

- How：Recognizes w according to production rules



Role of parser in a compiler

Parsing Method

- Top-Down Parsing
  - LL(1) parsing
    - Recursive descent parsing
    - Predictive parsing
- Bottom-Up Parsing
  - Operator-precedence parsing
  - LR parsing

Derive from start symbol to match input (leftmost derivation)

Reduce input to start symbol (inverse rightmost derivation)

Top-Down Parsing Example

For Grammer G[Z]	For Grammer G[S]
$Z \rightarrow aBd$	$S \rightarrow Ap \vert Bq$
$B \rightarrow d$	$A \rightarrow a \vert cA$
$B \rightarrow c$	$B \rightarrow b \vert dB$
$B \rightarrow bB$	
Derive the string **abcd**	Derive the string **ccap**

Bottom-Up Parsing Example

- Reduce from the terminal string to the grammar’ s start symbol

For Grammer G[Z]	For Grammer G[S]
$Z \rightarrow aBd$	$S \rightarrow Ap\|Bq$
$B \rightarrow d$	$A \rightarrow a\|cA$
$B \rightarrow c$	$B \rightarrow b\|dB$
$B \rightarrow bB$	
Reduction of string abcd	Reduction of string ccap

*

Outline

- Functions of a parser
- Overview of top-down parsing
- LL(1) parsing method
- Recursive descent parser
- Predictive parser

Top-Down Parsing

- Start from the grammar’s **start** symbol and **derive** downward
- Build a syntax tree and find the **leftmost** derivation

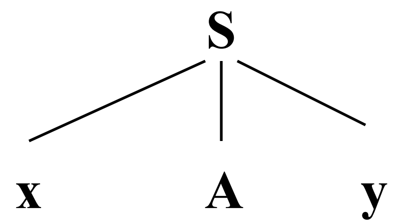
Example.Grammer G[S]: S→xAy, A→｜***

Determine if input string x * y is a sentence of G

x * y

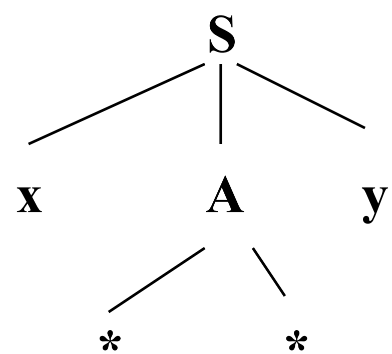
↑

S→xAy



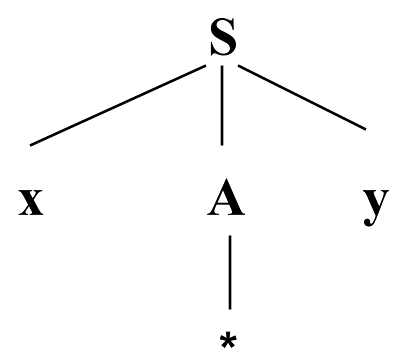
A→**

(Backtracking)



A→*

(Succeed)



Parsing Process:

$S \Rightarrow xAy$

$\Rightarrow x{*}{*}y$ **(Backtracking)**

$\Rightarrow x{*}y$ **(Succeed)**

*

Problems of Top-Down Parsing with Backtracking

- Left Recursion Problem
  - A grammar is left-recursive if ∃ nonterminal P such that $P \Rightarrow P\alpha$
  - Causes top-down parsing to fall into infinite loops
- False Matching Problem
- Backtracking
  - Consumes large amounts of time and space
  - Hard to locate the exact error position when parsing fails
  - Essentially exhaustive trial-and-error → low efficiency, high cost

Outline

- Functions of a parser
- Overview of top-down parsing
- LL(1) parsing method
- Recursive descent parser
- Predictive parser

LL(1) Parsing

- Scan input from Left to right, construct Leftmost derivation, look ahead 1 symbol at each step
- Purpose
  - Build a backtracking-free top-down parser
- Key Techniques
  - Eliminate left recursion
  - Eliminate backtracking (left factoring)
  - FIRST and FOLLOW sets
  - LL(1) Parsing Conditions
  - LL(1) Parsing Method

Key Techniques

- Eliminate left recursion
- Eliminate backtracking (left factoring)
- FIRST and FOLLOW sets
- LL(1) Parsing Conditions
- LL(1) Parsing Method

Left-Recursive Grammar

- A grammar is left-recursive if it has productions of the form

  a) Direct recursion

  $A \rightarrow A\beta \quad A \in V_N,\ \beta \in V^*$

  b) Indirect recursion

  $A \rightarrow B\beta$

  $B \rightarrow A\alpha \quad A,B \in V_N,\ \alpha \cdot \beta \in V^*$

Note: If a grammar is left-recursive, top-down parsing cannot be applied.

Example 1. Direct Left Recursion

$S \rightarrow Sa$; $S \rightarrow b$

Language: $L = \{ ba^n \mid n \geq 0 \}$

Example 2. Indirect Left Recursion

$A \rightarrow aB$

$A \rightarrow Bb$

$B \rightarrow Ac$

$B \rightarrow d$

Eliminating Direct Left Recursion

$P \rightarrow P\alpha \mid \beta$ ($\alpha \neq \varepsilon$, β does not start with P)

β

$\beta\alpha$

$\beta\alpha\alpha$

$\beta\alpha\alpha\alpha$

$P \rightarrow \beta P'$

$P' \rightarrow \alpha P' \mid \varepsilon$

… …

*

$E \to E+T \mid T$ $\quad$ $E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to T{*}F \mid F$ $\quad \Longrightarrow \quad$ $T \to FT'$

$T' \to {*}FT' \mid \varepsilon$

$F \to (E) \mid i$ $\quad$ $F \to (E) \mid i$

- General Case: Suppose productions for P are:

  $P \to P\alpha_1 \mid P\alpha_2 \mid \ldots \mid P\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

  where $\alpha_i \neq \varepsilon$，$\beta_i$ does not start with P，

  Rewrite as: $P \to \beta_1 P' \mid \beta_2 P' \mid \ldots \mid \beta_n P'$

  $P' \to \alpha_1 P' \mid \alpha_2 P' \mid \ldots \mid \alpha_m P' \mid \varepsilon$

Algorithm to Eliminate Left Recursion

(1) Arrange: P_1 、P_2 、… 、P_n

(2) Find & Eliminate:

FOR i := 1 TO n DO

BEGIN

FOR j:= 1 TO i -1 DO

Replace productions of the form $P_i \rightarrow P_j\gamma$ with :

$P_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

where $P_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all productions of P_j

Eliminate direct left recursion in P_i' s productions

END

(3) Simplify: Remove never-used productions

*

Eliminating Indirect Left Recursion (Example)

G(S) $S \to Qc|c \quad Q \to Rb|b \quad R \to Sa|a$

1) Arrange: S(1)、Q(2)、R(3)

2) Find: $S \to Q\,c\,|\,c$

$Q \to R\,b\,|\,b$

$R \to Rbca\,|\,bca\,|\,ca\,|\,a$

Eliminate direct left recursion :

$S \to Q\,c\,|\,c$

$Q \to R\,b\,|\,b$

$R \to bcaR'\,|\,caR'\,|\,aR'$

$R' \to bcaR'\,|\,\varepsilon$

Removing indirect left recursion is independent of nonterminal order

*

Eliminating Indirect Left Recursion (Exercise)

Grammar G(S)

$R \rightarrow Sa \mid a \qquad Q \rightarrow Rb \mid b \qquad S \rightarrow Qc \mid c$

Derivation: $S \Rightarrow Qc \Rightarrow Rbc \Rightarrow Sabc$, left recursion exists.

Order: R(1), Q(2), S(3)

$$
\begin{aligned}
&S \rightarrow Q\,c \mid c \\
&Q \rightarrow R\,b \mid b \\
&R \rightarrow bcaR' \mid caR' \mid aR' \\
&R' \rightarrow bcaR' \mid \varepsilon
\end{aligned}
$$

Exercise

- Eliminate left recursion from the following grammar

  $A \rightarrow aB$

  $A \rightarrow Bb$

  $B \rightarrow Ac$

  $B \rightarrow d$

Quiz-Canvas

- ch 4 Syntax Analysis - Left Recursion
- 2min

Key Techniques

- Eliminate left recursion
- Eliminate backtracking (left factoring)
- FIRST and FOLLOW sets
- LL(1) Parsing Conditions
- LL(1) Parsing Method

Is there a backtracking issue?

For the productions:

```
Statement → if Condition then Statement else Statement
          | while Condition do Statement
          | begin StatementList end
```

To parse a statement, the keywords **if**, **while**, **begin** indicate a unique alternative.

no backtracking required!

Is there a backtracking issue?

For the productions $S \rightarrow xAy \quad A \rightarrow **|*$

Sentences: $x*y$; $x**y$

Backtracking exists !

- If the current symbol = a, the next step is to expand A, and $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$. How to choose α_i?
  - If there is only one α_i starting with a, the replacement is unique.
  - If multiple α_i start with a, the replacement is not unique, backtracking is required.

Backtracking Solution

- Extract common left factors and transform the grammar so that the FIRST sets of all alternatives for any nonterminal are pairwise disjoint.

$$A \rightarrow \delta\beta_1 \mid \delta\beta_2 \mid \ldots \mid \delta\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

(Here γ_1、 γ_2、 …、 γ_m do not start with δ)

$$\Downarrow$$

$$A \rightarrow \delta A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

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

Example 1 G： $S \rightarrow aSb|aS|\varepsilon$

Extract common factors：

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

$$S \rightarrow \varepsilon$$

introduce a new symbol：

$$S \rightarrow aSA$$

$$A \rightarrow b|\varepsilon$$

$$S \rightarrow \varepsilon$$

Example 2 G： $S \rightarrow abc|abd|ae$

Extract $S \rightarrow a(bc|bd|e)$

Introduce $S \rightarrow aA$

$$A \rightarrow bc|bd|e$$

Extract more …

*

Advantages of No Backtracking

- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
  - For any nonterminal A, the first input symbol a uniquely determines the alternative α_i.
  - Success/failure of α_i fully represents A.
  - No trial or backtracking needed.

Key Techniques

- Eliminate left recursion
- Eliminate backtracking (left factoring)
- FIRST and FOLLOW sets
- LL(1) Parsing Conditions
- LL(1) Parsing Method

Grammar Requirements

- No left recursion.
- For each nonterminal, the FIRST sets of all alternatives are pairwise disjoint.

The FIRST set of a string α is defined as:

$$\mathrm{FIRST}(\alpha) = \{\, a \mid \alpha \overset{*}{\Rightarrow} a\ldots,\ a \in V_T \,\}$$

In particular, if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then $\varepsilon \in \mathrm{FIRST}(\alpha)$。

Condition (2) can be expressed as: for any nonterminal A, if $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ then $\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \Phi$, $i \neq j$

*

Computing the FIRST(X) Set

- For each grammar symbol X, compute FIRST(X):
  - If $X \in V_T$, FIRST(X)={X}
  - If $X \in V_N$, FIRST(X)={a|X→a…,$a \in V_T$}
  - If $X \in V_N$, and X →ε，则{ε} ∈ FIRST(X)
  - If $X \in V_N$, and $X \overset{*}{\rightarrow} Y_1 Y_2 \ldots Y_n$ ($Y_1 Y_2 \ldots Y_n \in V_N$)
    - If Y_1，Y_2，…，$Y_{i-1} \Rightarrow \varepsilon$，then FIRST($Y_1$)-{ε}, FIRST($Y_2$)-{ε}… FIRST($Y_{i-1}$)-{ε}, FIRST($Y_i$) ⊆ FIRST(X)
    - If $Y_i \Rightarrow \varepsilon (i=1,2\ldots n)$, then ε∈FIRST(X)

Question

G: $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid i$

Compute the FIRST Set for Each Nonterminal

Answer: FIRST (E) = FIRST (T)

= FIRST (F)

= { (, i }

FIRST (E') = { +, ε }

FIRST (T') = { *, ε}

Quiz-Canvas

- 3min
- ch4 Syntax Analysis - FIRST(X)

Constructing FIRST(α)

For a string $\alpha = X_1X_2\cdots X_n$，construct FIRST ($\alpha$)

（1） Init: FIRST(α) = FIRST (X_1) - {ε};

（2） If for all X_j ,$1<=j<= i -1$, ε∈FIRST (X_j), then add FIRST(X_j) -{ε} to FIRST(α) ；

（3） If for all X_j ,$1<= j <=n$, ε∈FIRST (X_j), then add

ε FIRST(α) 。

Is anything missing?

*

Question

G: $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid i$

Compute the FIRST set for the right-hand side of each production.

Answer

FIRST(TE') = { (, i }

FIRST(+TE') = { + }

FIRST(FT') = { (, i }

FIRST(*FT') = { * }

FIRST ((E)) = { (}

FIRST (i) = { i }

*

Exercise

- **Grammer G[S]**
  - S → aA|d
  - A → bS| ε
- For the input string **abd**, use the **FIRST(α)** method to derive the top-down parsing process.

  [Hint FIRST(aA)=a]

*

Key Techniques

- Eliminate left recursion
- Eliminate backtracking (left factoring)
- FIRST and FOLLOW sets
- LL(1) Parsing Conditions
- LL(1) Parsing Method

LL(1) Parsing Condition

Question: for a given input symbol ‘**a**’ and a nonterminal $\mathbf{A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n}$, $\mathrm{FIRST}(\alpha i) \cap \mathrm{FIRST}(\alpha j) = \emptyset$

if **a** $\boldsymbol{\notin}$ **FIRST($\boldsymbol{\alpha_i}$)** for all i,

➡ Does this mean there is no valid production to choose?

➡ Should the occurrence of ‘**a**’ be treated as a **syntax error** in the input?

Example

Let's check abd sentence

G(S):

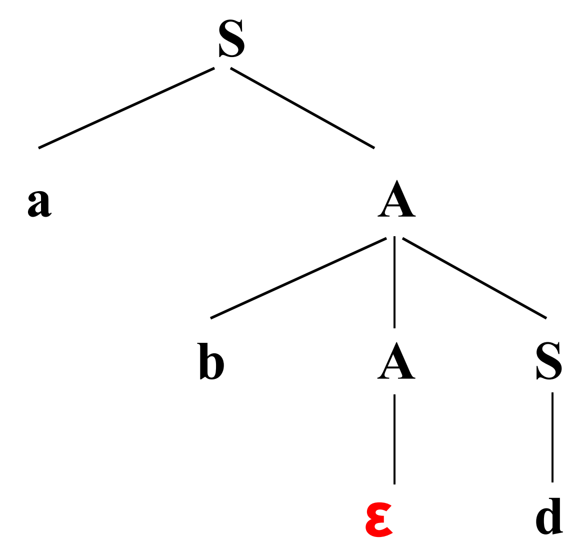
$S \rightarrow aA|d$

$A \rightarrow bAS|\varepsilon$

FIRST(S)={a, b}

FIRST(A) ={b, ε}

$$\begin{aligned} S &\Rightarrow aA \\ &\Rightarrow abAS \\ &\Rightarrow abS \\ &\Rightarrow abd \end{aligned}$$



This is because

(1) $A \rightarrow \varepsilon$, and (2) the following can be derived from the start symbol S: $S \Rightarrow \ldots Ad\ldots$ *

FOLLOW Set

Let S be the start symbol of grammar G. For any nonterminal A in G, define the FOLLOW set of A as: $\text{FOLLOW}(A) = \{ a \mid S \overset{*}{\Rightarrow} \ldots Aa \ldots,\ a \in V_T \}$

In Particular, if $S \overset{*}{\Rightarrow} \ldots A$, then

$\# \in \text{FOLLOW}(A)$

FOLLOW(A) contains all terminals or "#" that can appear immediately after A in any sentential form.

LL(1) Grammar Conditions — Refinement

- When a non-terminal A faces an input symbol a, and $a \notin FIRST(\alpha_i)$ (for any i), if some candidate first set of A contains ϵ (i.e., $\epsilon \in FIRST(A)$), then if $a \in FOLLOW(A)$,A can match automatically (i.e., choose $A \rightarrow \epsilon$). Otherwise, the appearance of a is considered a syntax error.
- To perform syntax analysis without backtracking, the third condition that the grammar must satisfy is:
  - $FIRST(A) \cap FOLLOW(A) = \emptyset$

Construction of $FOLLOW(A)$

For each non-terminal A in the grammar G, the method to construct $FOLLOW(A)$ is:

（1）If A is the start symbol of the grammar, add # to $FOLLOW(A)$.

(2)If there is a production $B \rightarrow \alpha A \beta$, add $FIRST(\beta) - \{\epsilon\}$ to $FOLLOW(A)$.

(3)If there is a production $B \rightarrow \alpha A$ or ($B \xrightarrow{*} \alpha A \beta$ and $\beta \Rightarrow \epsilon$),
add $FOLLOW(B)$ to $FOLLOW(A)$.

(4)Repeat the above rules until $FOLLOW(A)$ no longer increases.

Example. G: $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid i$

Find the $FOLLOW$ set for each non-terminal.

FOLLOW (E) = { #,)}

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

FOLLOW (T) = { +, #,) }

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

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

Quiz-Canvas

- 5min
- ch4 Syntax Analysis -FOLLOW(X)

Quiz

- **Grammer G[S]**
  - **S → aA|d**
  - **A → bAS| ε**
- For the input string "abd", use the FIRST(α) + FOLLOW(A) method to derive the top-down parsing process.

[Hint FIRST(aA)=a]

LL (1) Grammer

- If the grammar G satisfies the following conditions:
  - The grammar eliminates left recursion;
  - For each non-terminal A, the FIRST sets of the right-hand sides of its productions are disjoint, i.e.,
    - If $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, *then* $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$ *for* $i \neq j$;
  - For each non-terminal A in the grammar, if some production of A contains ϵ in its FIRST set, then $FIRST(A) \cap FOLLOW(A) = \emptyset$;
- Then the grammar G is called an **LL(1) grammar**.

*

Key Techniques

- Eliminate left recursion
- Eliminate backtracking (left factoring)
- FIRST and FOLLOW sets
- LL(1) Parsing Conditions
- LL(1) Parsing Method

LL(1) Parsing Method

- For an LL(1) grammar, an effective top-down, backtracking-free analysis can be performed for a given input string.
- Suppose the current input symbol is a, and we need to match it using the non-terminal A, where $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ .The analysis can be performed as follows:
  - If $\boldsymbol{a} \in \boldsymbol{FIRST}(\boldsymbol{\alpha_i})$,assign $\boldsymbol{\alpha_i}$ to perform the matching task.
  - Otherwise:
    - If $\boldsymbol{\epsilon} \in \boldsymbol{FIRST}(\boldsymbol{A})$ and $\boldsymbol{a} \in \boldsymbol{FOLLOW}(\boldsymbol{A})$,allow $\boldsymbol{A}$ to automatically match with $\boldsymbol{\epsilon}$.
    - Otherwise, the appearance of $\boldsymbol{a}$ is a syntax error.

Quiz-Canvas

- ch4 Syntax Analysis 4 - LL(1) Grammar

LL(1) Parsing Method

- ✓ Recursive descent parser/递归下降分析程序
- predictive parser/预测分析程序

Recursive descent parser

- **Conditions**
  - Satisfy the conditions of the above LL(1) grammar.
- **Composition**
  - A set of recursive processes.
  - Each recursive process corresponds to a non-terminal of G.
- **Basic idea**
  - Start from the **start symbol** of the grammar.
  - Perform syntax analysis under the control **of grammar rules**.
  - Scan characters of the source program, when encountering a syntax component A, **call the subroutine to analyze A.**

Recursive descent parser

- For each non-terminal A, write a corresponding subroutine $P(A)$;
- For the rule $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$, the corresponding subroutine $P(A)$ is constructed as follows:

IF ch IN FIRST(α_1) THEN P(α_1)

ELSE IF ch IN FIRST(α_2) THEN P(α_2)

ELSE ……

ELSE IF ch IN FIRST(α_n) THEN P(α_n)

ELSE IF ($\varepsilon \in$FIRST(A))AND (ch IN FOLLOW(A))

THEN RETURN

ELSE ERROR

- For the symbol string $\alpha = \gamma_1\gamma_2\gamma_3 \ldots \gamma_m$,the corresponding subroutine $P(\alpha)$ is:

```
BEGIN    P (γ1 )
         P (γ2 )
          …
         P (γm)
END
```

- If $\gamma_i \in V_T$,then $P(\gamma_i)$ is:

  IF ch= γi THEN read(ch) ELSE ERROR ;

- If $\gamma_i \in V_N$,then $P(\gamma_i)$ is the corresponding subroutine from above.

$E \to TE'$; $E' \to +TE' \mid \varepsilon$; $T \to FT'$; $T' \to *FT' \mid \varepsilon$; $F \to (E) \mid i$

FIRST(+ TE ')={ + }　　FIRST(*FT ')={ * }

FOLLOW(E ')={),# }　　FOLLOW(T ')={ +,),# }

FIRST((E))={ (}　　FIRST(i)={ i }

```
P( E )
BEGIN
   P(T); P(E')
END;
```

```
P( T)
BEGIN
   P(F); P(T')
END;
```

```
P(E')
IF  ch =" + "  THEN
BEGIN
      read(ch);P(T);P(E');
END;
ELSE IF (ch =" )"  OR
ch='#') THEN
     return;
ELSE ERROR;
```

```
P(T')
IF  ch=' *'THEN
  BEGIN
      read(ch);P(F);P(T');
  END;
ELSE IF (ch='+'OR
ch=')'OR ch='#')THEN
     return;
ELSE ERROR;
```

```
P( F)
IF  ch='i' THEN read(ch);
ELSE  IF  ch =  '('  THEN
BEGIN
      read(ch);P(E);
      IF ch =')' THEN read(ch);
      ELSE  ERROR
END
ELSE  ERROR;
```

Exercise

- P81,1 用递归下降分析程序书写语法分析器

G'（S）：S→a|∧|（T）

T→ST'

T'→，ST'|ε

LL(1) Parsing Method

- Recursive descent parser/递归下降分析程序
- ✓ predictive parser/预测分析程序

*

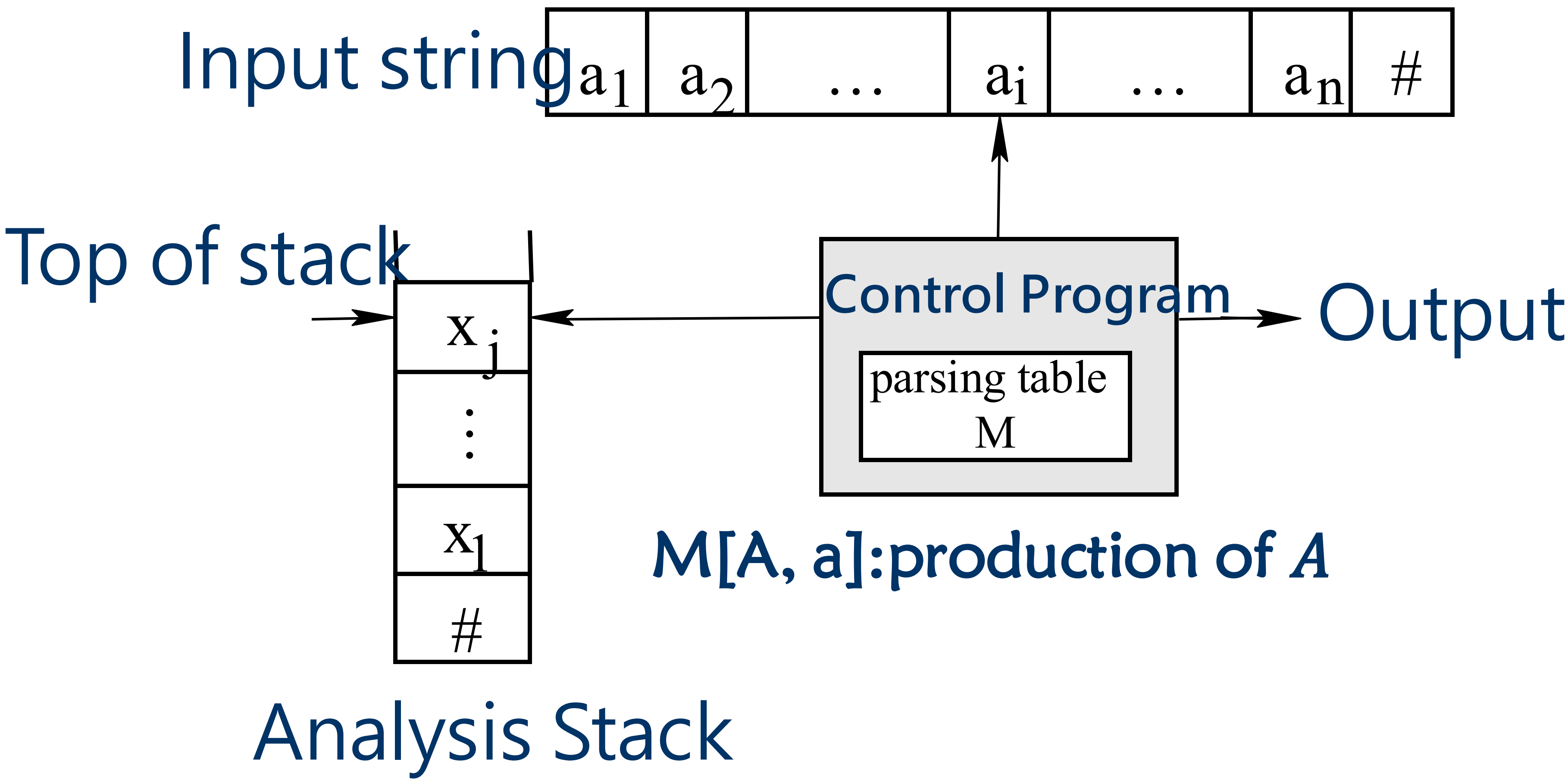
Predictive Parsing

- **Limitations of Recursive Descent Parser**
  - Requires a language and compiler system capable of supporting recursive processes.
- Predictive Parsing Program
  - Uses a LL(1) parsing table and symbol stack for joint control
  - Effective method

Predictive Parsing

- **Basic Idea of Predictive Parsing Program**
  - Select a production based on the current input symbol
  - If it matches the first symbol of the derivation, move to the next input symbol
  - Continue until the input string is fully parsed
- **Components of LL(1) Predictive Parser**
  - **LL(1) Parsing Table** (Prediction Table)
  - **Symbol Stack** (Last-In-First-Out)
  - **Control Program** (Table-Driven Program)

*



Predictive Parsing Program

*

Example

- **Grammer A**

$$A \rightarrow aB$$

$$B \rightarrow b$$

- **Grammer A**

$$A \rightarrow aB$$

$$B \rightarrow b|\varepsilon$$

LL(1) Parsing Table

- If the grammar has m non-terminals and n terminals, the LL(1) parsing table is a matrix M of size $(m+1) \times (n+2)$.
  - The row headers are the grammar's non-terminals.
  - The column headers are the terminals and the end-of-input symbol #.
  - $M[A, a]$ is a production for A, indicating the production to use when A faces a, or a blank (error flag).

*

$E \to TE'$; $E' \to +TE' \mid \varepsilon$;

$T \to FT'$; $T' \to *FT' \mid \varepsilon$;

$F \to (E) \mid i$

LL(1) parsing table

	i	+	*	(	)	#
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to i$			$F \to (E)$		

Analysis Stack

- The STACK stores the grammar symbols during the parsing process.
  - At the start of the analysis, place a "#" at the bottom of the stack, followed by the start symbol of the grammar.
  - The analysis is successful when only "#" remains in the stack and the input pointer points to the end-of-input symbol "#".

Control Program

- The main control program decides the parser's actions based on the top stack symbol x and the current input symbol a:
  - If $x = a = \#$, the analysis is successful.
  - If $x = a \neq \#$, pop x from the stack, move the input pointer, and read the next symbol.
  - If x is a non-terminal A, look up $M[A, a]$:
    - If $M[A, a]$ is a production, pop A and push the right-hand side in reverse order.
    - If $M[A, a]$ is $A \rightarrow \epsilon$, pop A.
    - If $M[A, a]$ is empty, call the error handling program.

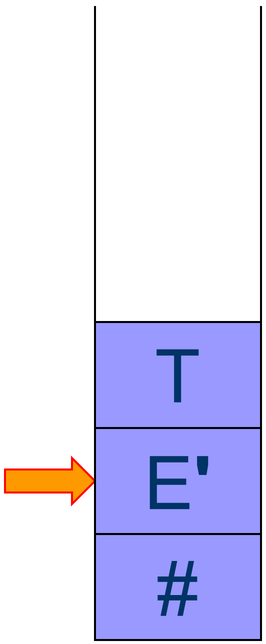
*

The Pseudocode for the Main Control Program

```
BEGIN
    push('#'); push('S');  // Push # and start symbol S onto the stack
    read the first input symbol into a;
    FLAG := TRUE;

    WHILE FLAG DO
    BEGIN
        X := pop();  // Pop the top symbol from the stack
        IF X ∈ VT THEN
            IF X = a THEN
                read the next input symbol into a;
            ELSE
                ERROR;
        ELSE IF X = "#" THEN
            IF X = a THEN
                FLAG := FALSE;
            ELSE
                ERROR;
        ELSE IF M[X, a] = {X → X1 … Xk} THEN
            push Xk, Xk-1, … , X1 onto the stack (reverse order);
        ELSE
            ERROR;
    END WHILE;
END
```

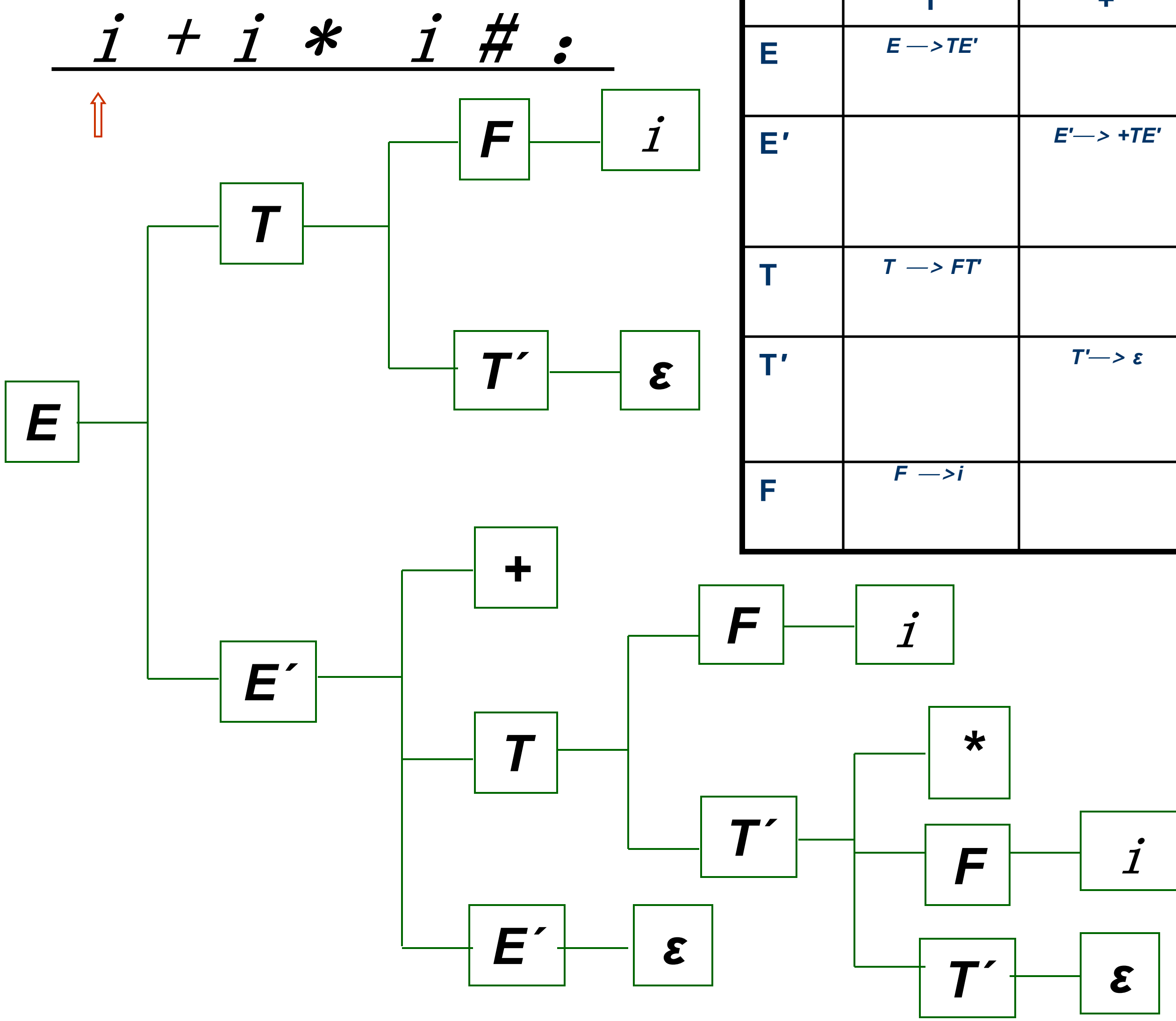
i + i * i #



P80

	i	+	*	(	)	#
E	$\boxed{E \rightarrow TE'}$			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T' →ε	T' →*FT'		T' →ε	T' →ε
F	F→i			F→(E)		

The syntax tree generated by the above analysis process



	i	+	*	(	)	#
E	E —>TE'			E —> TE'		
E'		E'—> +TE'			E'—> ε	E'—> ε
T	T —> FT'			T —> FT'		
T'		T'—> ε	T'—> *FT'		T'—> ε	T'—> ε
F	F —>i			F —>(E)		

*

$E \to TE'$; $E' \to +TE' \mid \varepsilon$;
$T \to FT'$; $T' \to *FT' \mid \varepsilon$;
$F \to (E) \mid i$

FOLLOW(E ')={),# }
FOLLOW(T ')={ +,),# }

i + i * i #

	Stack	Input	Production
0	#E	i+i*i#	
1	#E′T	i+i*i#	$E \to TE'$
2	#E′T′F	i+i*i#	$T \to FT'$
3	#E′T′i	i+i*i#	$F \to i$
4	#E′T′	+i*i#	
5	#E′	+i*i#	$T' \to \varepsilon$
6	#E′T+	+i*i#	$E' \to +TE'$
7	#E′T	i*i#	
8	#E′T′F	i*i#	$T \to FT'$
9	#E′T′i	i*i#	$F \to i$
10	#E′T′	*i#	
11	#E′T′F*	*i#	$T' \to *FT'$
12	#E′T′F	i#	
13	#E′T′i	i#	$F \to i$
14	#E′T′	#	
15	#E′	#	$T' \to \varepsilon$
16	#	#	$E' \to \varepsilon$

Conclusion

- The output productions are from the **leftmost derivation**. The stack holds **the right-hand side of productions**, waiting to match with a.
- When the top non-terminal X faces a string starting with a, the **parsing table** indicates how to expand the syntax tree, and **errors are detected immediately**.
- **Features**:
  - **Stack**: Sentence parts, right-hand side of productions, and undetermined symbols.
  - **Table**: Guides expansions of non-terminals based on terminals.

Construction of LL(1) Parsing Table

In a predictive parsing program, except for the **parsing table**, which **differs** depending on the grammar, **the analysis stack and control program** remain the **same**. Therefore, constructing a predictive parsing program is essentially the same as constructing the LL(1) parsing table for the grammar.

- **Questions**:
  - Where should the productions be placed in the table?
  - Divide the productions for A into two types:
    - One type: $A \to a \ldots$
    - The other type: $A \to \epsilon$

*

Construction of LL(1) Parsing Table

For each production $A \to \alpha$, perform:

If $a \in FIRST(\alpha)$,set $M[A, a] = A \to \alpha$

If $\epsilon \in FIRST(A)$,for $b \in FOLLOW(A)$,set $M[A, b] = A \to \epsilon$

For all other cases, set $M[A, a] = ERROR$

*

$E \rightarrow TE'$; $E' \rightarrow +TE' \mid \varepsilon$; $T \rightarrow FT'$; $T' \rightarrow *FT' \mid \varepsilon$; $F \rightarrow (E) \mid i$

FIRST(TE')={ (, i }

FIRST(FT')={ (, i }

FIRST((E))={ (}

FOLLOW(E ')={),# }

FIRST(+ TE')={ + }

FIRST(*FT ')={ * }

FIRST(i)={ i }

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

	i	+	*	(	)	#
E						
E'						
T						
T'						
F						

The parsing table and LL(1) grammar

- If the predictive parsing table $M[A, a]$ for grammar G does not contain multiple definitions for any entry, then and only then is G an LL(1) grammar.
- For grammar G to be LL(1), for every non-terminal A and any two distinct productions $A \rightarrow \alpha \mid \beta$, the following conditions must hold:

*

$$FIRST(\alpha) \cap FIRST(\beta) = \emptyset$$

If $A \Rightarrow \epsilon$, then $FIRST(A) \cap FOLLOW(A) = \emptyset$ *

Conclusion

- Eliminating Left Recursion
- Eliminating Backtracking: Common left factor extraction method
- Recursive Descent Parser
- Predictive Parser
  - LL(1) Parsing Table
    - FIRST(α)
    - FOLLOW(X)

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

감사합니다
Korean

Tack så mycket
Swedish

धन्यवाद
Hindi

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

Dankon
Esperanto

谢谢！

Thank You
English

ありがとうございます
Japanese

Trugarez
Breton

Danke
German

Tak
Danish

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุ
ณ
Thai

go raibh maith agat
Gaelic

*