南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 3: Syntax Analysis

Yepang Liu
liuyp1@sustech.edu.cn

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
  - Recursive-descent parsing
  - Non-recursive predictive parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

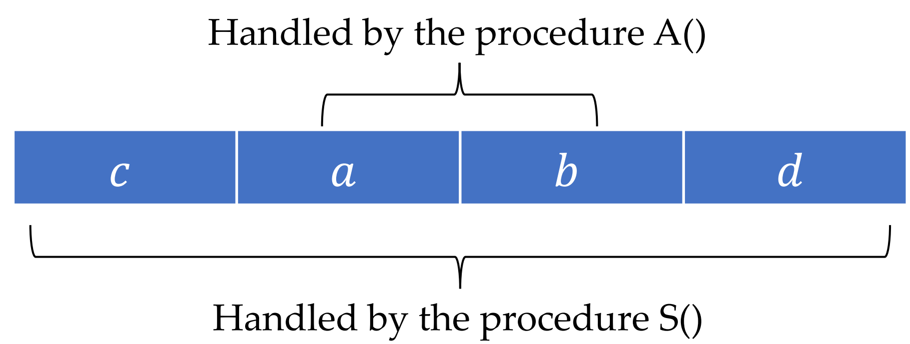
Recursive-Descent Parsing (递归下降的语法分析)

- A recursive-descent parsing program has a set of procedures, one for each nonterminal
  - The procedure for a nonterminal deals with a substring of the input
- Execution begins with the procedure for the start symbol
  - Announce success if the procedure scans the entire input (the start symbol derives the whole input via applying a series of productions)

CFG:

$$\boxed{\begin{array}{l} S \rightarrow cAd \\ A \rightarrow ab \end{array}}$$

Input string:



A Typical Procedure for A Nonterminal

```
void A() {
        Choose an A-production, A → X1X2 ··· Xk;
        for ( i = 1 to k ) {
              if ( Xi is a nonterminal )
                    call procedure Xi();
              else if ( Xi equals the current input symbol a )
                    advance the input to the next symbol;
              else /* an error has occurred */;
        }
}
```

Predict

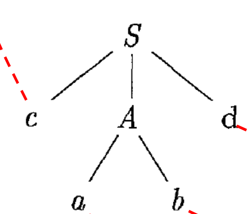
Match

CFG:

$S \rightarrow cAd$
$A \rightarrow ab$

Parsing input:

c	*a*	*b*	*d*



call S() → match "c"

→ call A() → match "ab" → A() return

→ match "d" → S() return

Backtracking (回溯)

```
    void A() {
1)      Choose an A-production, A → X_1 X_2 ··· X_k;
2)      for ( i = 1 to k ) {
3)          if ( X_i is a nonterminal )
4)              call procedure X_i();
5)          else if ( X_i equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

If there is a failure at line 7, does this mean that there must be syntax errors?

Backtracking (回溯)

```
   void A() {
1)     Choose an A-production, A → X1 X2 ··· Xk;
2)     for ( i = 1 to k ) {
3)         if ( Xi is a nonterminal )
4)             call procedure Xi();
5)         else if ( Xi equals the current input symbol a )
6)             advance the input to the next symbol;
7)         else /* an error has occurred */;
       }
   }
```

The failure might be caused by a wrong choice of *A*-production at line 1 !!!

Backtracking (回溯)

- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
void A() {
      Choose an A-production, A → X1 X2 ··· Xk;
      for ( i = 1 to k ) {
            if ( Xi is a nonterminal )
                  call procedure Xi();
            else if ( Xi equals the current input symbol a )
                  advance the input to the next symbol;
            else /* an error has occurred */;
      }
}
```

Instead of exploring one *A*-production, we must try each possible production in some order.

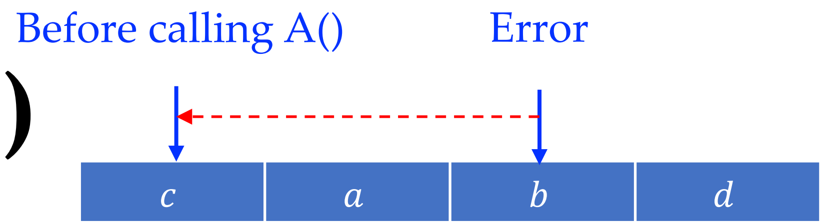
Backtracking (回溯)

- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

When there is a failure at line 7, return to line 1 and try another *A*-production.

Backtracking (回溯)



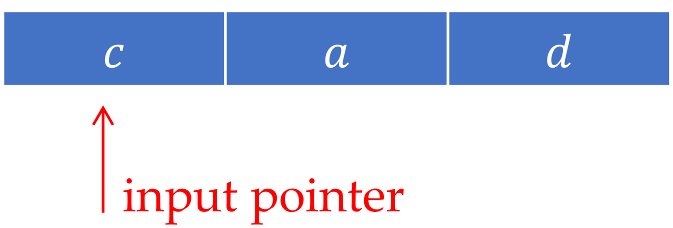
- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

In order to try another *A*-production, we must reset the input pointer that points to the next symbol to scan (failed trials consume symbols)

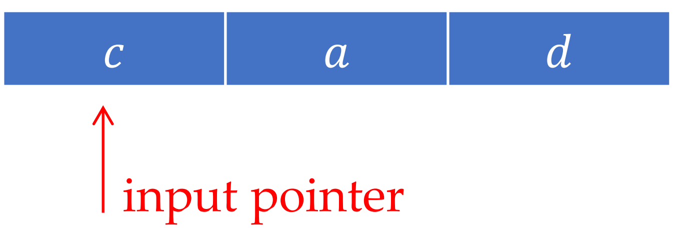
Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid \boxed{a}$ One more production for A
- Input string: cad

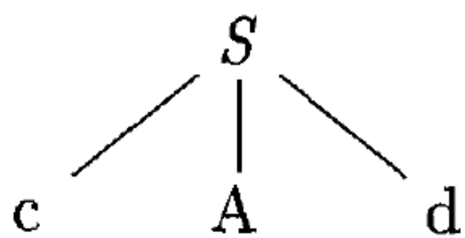


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

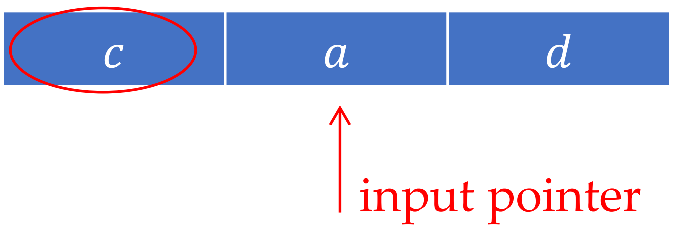


– S has only one production, apply it

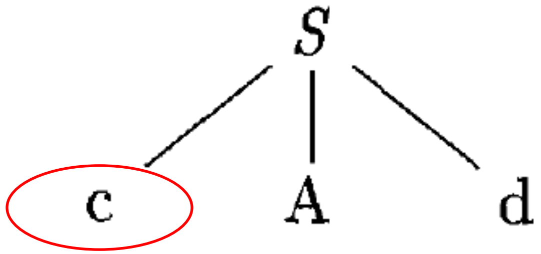


Backtracking Example

- Grammar: $S \to cAd \quad A \to ab \mid a$
- Input string: cad

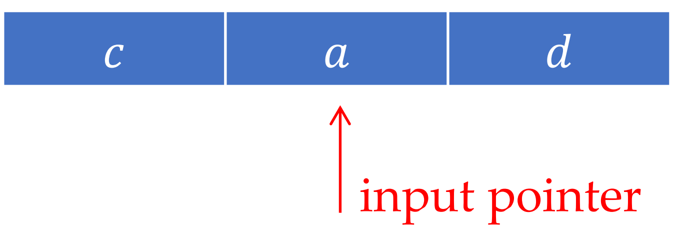


- The leftmost leaf matches c in input
- Advance input pointer

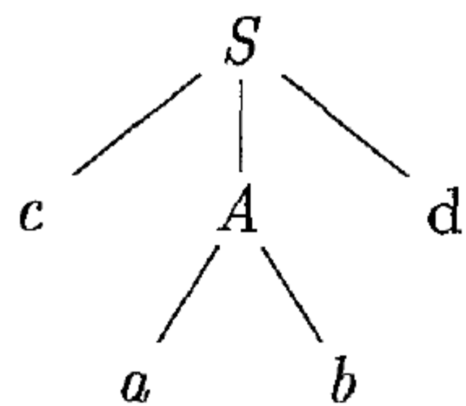


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: *cad*

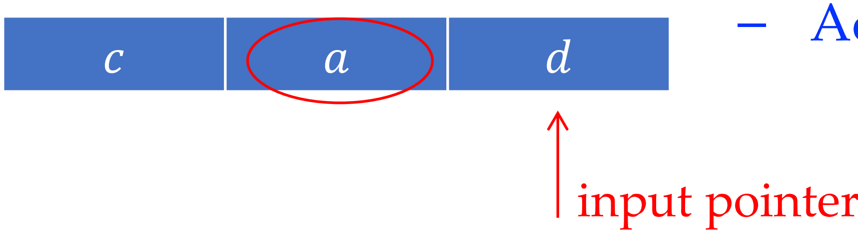


– Expand *A* using the first production

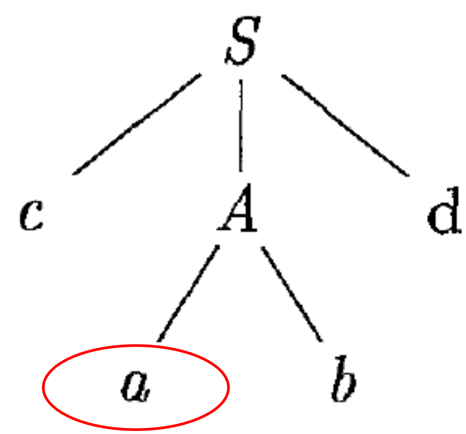


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: *cad*

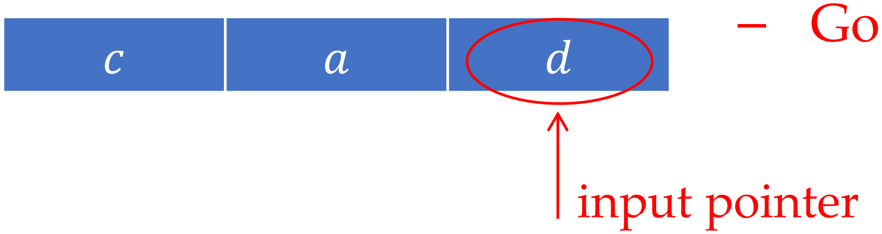


- Leftmost leaf matches a in input
- Advance input pointer

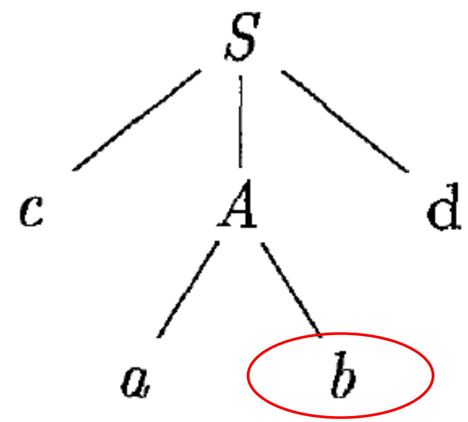


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: *cad*

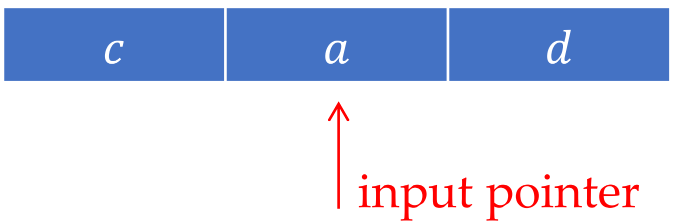


- Symbol mismatch
- Go back to try another *A*-production

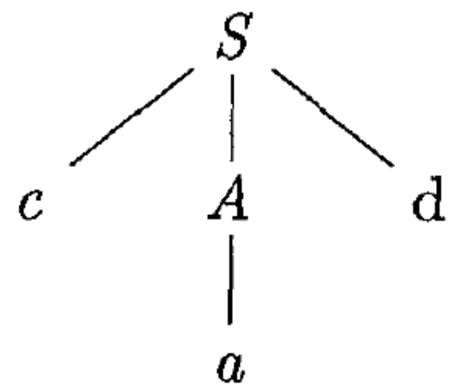


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

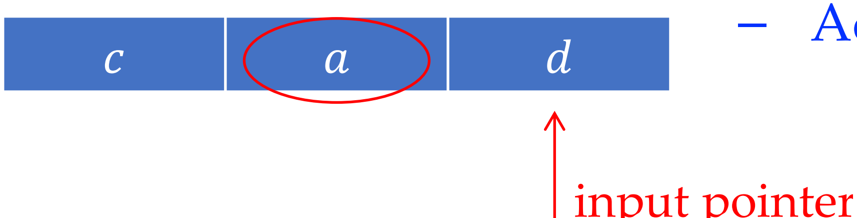


- Reset input pointer
- Expand A using its second production

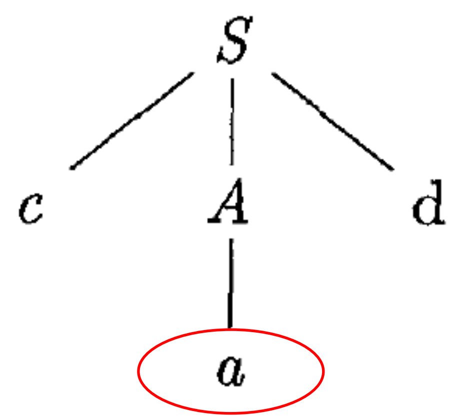


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad



- Leftmost leaf matches a in input
- Advance input pointer

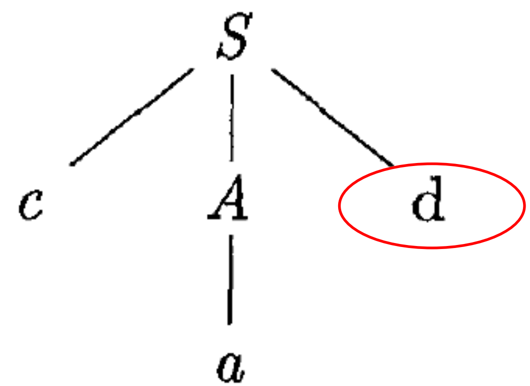


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

- The last leaf node matches d in input
- Announce success!

Scanned entire input



The Problem of Left Recursion

Suppose there is only one A-production, $\boldsymbol{A \rightarrow A\alpha}$ ...

void $A()$ {
    Choose an A-production, $\boxed{A \rightarrow X_1 X_2 \cdots X_k}$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

Recursively rewriting A without matching any terminals

If there is left recursion in a CFG, a recursive-descent parser may go into an infinite loop! Revise the CFG before parsing!!!

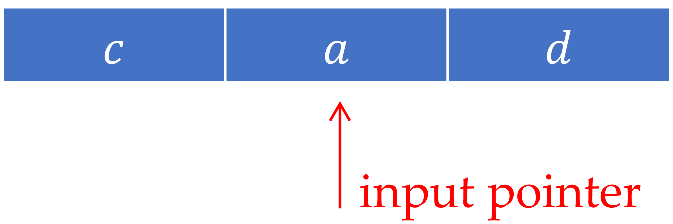
Can We Avoid Backtracking?

void $A()$ {
1) Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
2) **for** ($i = 1$ to k) {
3) **if** (X_i is a nonterminal)
4) call procedure $X_i()$;
5) **else if** (X_i equals the current input symbol a)
6) advance the input to the next symbol;
7) **else** /* an error has occurred */;
}
}

Key problem: At line 1, we make ***random choices*** (brute force search)

Can We Avoid Backtracking?

- Grammar: $S \rightarrow cAd \quad A \rightarrow c \mid a$
- Input string: cad



When rewriting A, is it a good idea to choose $A \rightarrow c$?

No! If we look ahead, the next char in the input is a.
$A \rightarrow c$ is obviously a bad choice!!!

Looking Ahead Helps!

- Suppose the input string is $\boldsymbol{x}\boldsymbol{a}\ldots$
- Suppose the current sentential form is $\boldsymbol{xA\beta}$
  - $\boldsymbol{A}$ is a non-terminal; $\boldsymbol{\beta}$ may contain both terminals and non-terminals

If we know the following fact for the productions $\boldsymbol{A \rightarrow \alpha \mid \gamma}$:

- $a \in FIRST(\alpha)$: α derives strings that begin with a
- $a \notin FIRST(\gamma)$: γ derives strings that do not begin with a

*$FIRST(\alpha)$ denotes the set of beginning terminals of strings derived from α

After matching x, which production should we choose to rewrite A?

$\boldsymbol{A \rightarrow \alpha}$

Computing FIRST

- **FIRST(X)**, where X is a grammar symbol
  - If X is a terminal, then FIRST(X) = {X}
  - If X is a nonterminal and $X \to \epsilon$, then add ϵ to FIRST(X)
  - If X is a nonterminal and $X \to Y_1 Y_2 \dots Y_k$ ($k \geq 1$) is a production
    - If for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1), …, FIRST(Y_{i-1}), then add a to FIRST(X)
    - If ϵ is in all of FIRST(Y_1), …, FIRST(Y_k), then add ϵ to FIRST(X)

Computing FIRST Cont.

- **FIRST($X_1X_2 \ldots X_n$)**, where $X_1X_2 \ldots X_n$ is a string of grammar symbols
  - Add all non-ϵ symbols of FIRST(X_1) to FIRST($X_1X_2 \ldots X_n$)
  - If ϵ is in FIRST(X_1), add non-ϵ symbols of FIRST(X_2) to FIRST($X_1X_2 \ldots X_n$)
  - If ϵ is in FIRST(X_1) and FIRST(X_2), add non-ϵ symbols of FIRST(X_3) to FIRST($X_1X_2 \ldots X_n$)
  - …
  - If ϵ is in FIRST(X_i) for all i, add ϵ to FIRST($X_1X_2 \ldots X_n$)

FIRST Example

- **Grammar**
  - $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \mathbf{id}$
- **FIRST sets**
  - $\text{FIRST}(F) = \{(, \mathbf{id}\}$
  - $\text{FIRST}(T) = \text{FIRST}(F) = \{(, \mathbf{id}\}$
  - $\text{FIRST}(E) = \text{FIRST}(T) = \{(, \mathbf{id}\}$
  - $\text{FIRST}(E') = \{+, \epsilon\}$ $\quad \text{FIRST}(T') = \{*, \epsilon\}$
  - $\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \mathbf{id}\}$
  - …

Strings derived from F or T must start with (or **id**

Looking Ahead Helps Cont.

- Suppose the input string is $\boldsymbol{x}\boldsymbol{a}\ldots$
- Suppose the current sentential form is $\boldsymbol{x}\boldsymbol{A}\boldsymbol{\beta}$
  - $\boldsymbol{A}$ is a non-terminal; $\boldsymbol{\beta}$ may contain both terminals and non-terminals

If we know that for the production $\boldsymbol{A} \rightarrow \boldsymbol{\alpha}$, $\epsilon \in FIRST(\alpha)$, can we choose the production to rewrite A?

Yes, only if β can derive strings beginning with a, that is, A can be followed by a in some sentential forms (i.e., $\boldsymbol{a} \in \boldsymbol{FOLLOW}(\boldsymbol{A})$)

Computing FOLLOW

- **Computing FOLLOW(A) for all nonterminals A**
  - Add \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker
  - Apply the rules below, until all FOLLOW sets do not change
    1. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B)
    2. If there is a production $A \rightarrow \alpha B$ (or $A \rightarrow \alpha B \beta$ and FIRST(β) contains ϵ) then everything in FOLLOW(A) is in FOLLOW(B)

By definition, ϵ will not appear in any FOLLOW set

FOLLOW Example

- **Grammar**
  - $E \to TE'$ $\quad E' \to +TE' \mid \epsilon$
  - $T \to FT'$ $\quad T' \to * FT' \mid \epsilon$ $\quad F \to (E) \mid \textbf{id}$

- **FOLLOW sets**
  - $\boxed{\text{FOLLOW}(E) = \{\$,)\}}$
  - $\text{FOLLOW}(E') = \{\$,)\}$
  - $\boxed{\text{FOLLOW}(T) = \{+, \$,)\}}$
  - $\text{FOLLOW}(T') = \{+, \$,)\}$
  - $\text{FOLLOW}(F) = \{*, +, \$,)\}$

- \$ is always in FOLLOW(E)
- Everything in FIRST()) except ϵ is in FOLLOW(E)

- Everything in FIRST(E') except ϵ is in FOLLOW(T)
- Since $E' \to \epsilon$, everything in FOLLOW(E) and FOLLOW(E') is in FOLLOW(T)

A Quick Summary

Why Do We Compute FIRST & FOLLOW?

- For a production $head \rightarrow body$, when we are trying to rewrite $head$, if we know $FIRST(body)$, that is, what terminals can strings derived from body start with, we can decide whether to choose $head \rightarrow body$ by looking at the next input symbol.
  - If the next input symbol is in $FIRST(body)$, the production may be a good choice.

- For a production $head \rightarrow \epsilon$ (or $head$ can derive ϵ in some steps), when we are trying to rewrite $head$, if we know $FOLLOW(head)$, that is, what terminals can follow $head$ in various sentential forms, we can decide whether to choose $head \rightarrow \epsilon$ by looking at the next input symbol.
  - If the next input symbol is in $FOLLOW(head)$, the production may be a good choice.

LL(1) Grammars

- Recursive-descent parsers needing no backtracking can be constructed for a class of grammars called **LL(1)**
  - 1^{st} L: scanning the input from left to right
  - 2^{nd} L: producing a leftmost derivation
  - 1: using one input symbol of lookahead at each step to make parsing decision

LL(1) Grammars Cont.

A grammar G is LL(1) if and only if for any two distinct productions $A \to \alpha \mid \beta$, the following conditions hold:

1. There is no terminal a such that α and β derive strings beginning with a
2. At most one of α and β can derive the empty string
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in $\mathtt{FOLLOW}(A)$ and vice versa

* The three conditions rule out the possibility of applying both productions so that there is a unique choice of production at each "predict" step by looking at the next input symbol

More formally:

1. $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$ (conditions 1-2 above)
2. If $\epsilon \in \mathrm{FIRST}(\beta)$, then $\mathrm{FIRST}(\alpha) \cap \mathtt{FOLLOW}(A) = \emptyset$ and vice versa

LL(1) Grammars Cont.

- For LL(1) grammars, during recursive-descent parsing, the proper production to apply for a nonterminal can be selected by looking only at the current input symbol

Grammar: $stmt \rightarrow \underbrace{\textbf{if}(\textbf{expr})\ stmt\ \textbf{else}\ stmt}_{1} \mid \underbrace{\textbf{while}(\textbf{expr})\ stmt}_{2} \mid \underbrace{\textbf{a}}_{3}$

Parsing steps for input: **if(expr) while(expr) a else a**

1. Rewrite the start symbol *stmt* with ①: **if(expr)** *stmt* **else** *stmt*
2. Rewrite the leftmost *stmt* with ②: **if(expr) while(expr)** *stmt* **else** *stmt*
3. Rewrite the leftmost *stmt* with ③: **if(expr) while(expr) a else** *stmt*
4. Rewrite the leftmost *stmt* with ③: **if(expr) while(expr) a else a**

Parsing Table (预测分析表)

- We can build parsing tables for recursive-descent parsers (**LL parsers**)
- A predictive **parsing table** is a two-dimensional array that determines which production the parser should choose when it sees a nonterminal A and a symbol a on its input stream
- The parsing table of an LL(1) parser has no entries with multiple productions

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Parsing Table Construction

The following algorithm can be applied to any CFG

- **Input:** Grammar G **Output:** Parsing table M
- **Method:**
  - For each production $A \rightarrow \alpha$ of G, do the following:
    - For each terminal a in $\mathrm{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
    - If ϵ is in $\mathrm{FIRST}(\alpha)$, then for each terminal b (including the right endmarker \$) in $\mathrm{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$
  - Set all empty entries in the table to **error**

Fill the table entries so that when rewriting A, we know what production to choose by checking the next input symbol

Parsing Table Construction Example

- **Grammar**
  - $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \mathbf{id}$
- **FIRST sets:** E, T, F: $\{(, \mathbf{id}\}$ $\quad E'$: $\{+, \epsilon\}$ $\quad T'$: $\{*, \epsilon\}$
- **FOLLOW sets:** E, E': $\{\$,)\}$ $\quad T, T'$: $\{+, \$,)\}$ $\quad F$: $\{*, +, \$,)\}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$\boxed{E \rightarrow TE'}$			$\boxed{E \rightarrow TE'}$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

For $E \rightarrow TE'$:

FIRST(TE')

= FIRST(T)

= $\{(, \mathbf{id}\}$

Parsing Table Construction Example

- **Grammar**
  - $E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $T' \rightarrow * FT' \mid \epsilon$ $F \rightarrow (E) \mid \textbf{id}$
- **FIRST sets:** E, T, F: {(, **id**} E': $\{+, \epsilon\}$ T': $\{*, \epsilon\}$
- **FOLLOW sets:** E, E': {\$,)} T, T': {+, \$,)} F: {∗, +, \$,)}

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$\boxed{E' \rightarrow \epsilon}$	$\boxed{E' \rightarrow \epsilon}$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

For $E' \rightarrow \epsilon$:

ϵ in FIRST(ϵ)

FOLLOW(E')

= {\$,)}

Conflicts in Parsing Tables

- **Grammar:** $S \to \mathbf{i}E\mathbf{t}SS' \mid \mathbf{a}$ $\boxed{S' \to \mathbf{e}S \mid \epsilon}$ $E \to \mathbf{b}$
  - FIRST($\mathbf{e}S$) = {**e**}, so we add $S' \to \mathbf{e}S$ to $M[S', \mathbf{e}]$
  - FOLLOW(S') = {\$, **e**}, so we add $S' \to \epsilon$ to $M[S', \mathbf{e}]$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

- LL(1) grammar is never ambiguous.
- This grammar is not LL(1). The language has no LL(1) grammar !!!

Recursive-Descent Parsing for LL(1) Grammars

```
   void A() {
1)     Choose an A-production, A → X1 X2 ··· Xk;
2)     for ( i = 1 to k ) {
3)         if ( Xi is a nonterminal )
4)             call procedure Xi();
5)         else if ( Xi equals the current input symbol a )
6)             advance the input to the next symbol;
7)         else /* an error has occurred */;
       }
   }
```

Replace line 1 with: Choose A-production according to the parse table

- Assume input symbol is a, then the choice is the production in $M[A, a]$

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
  - Recursive-descent parsing
  - Non-recursive predictive parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Recall Recursive-Descent Parsing

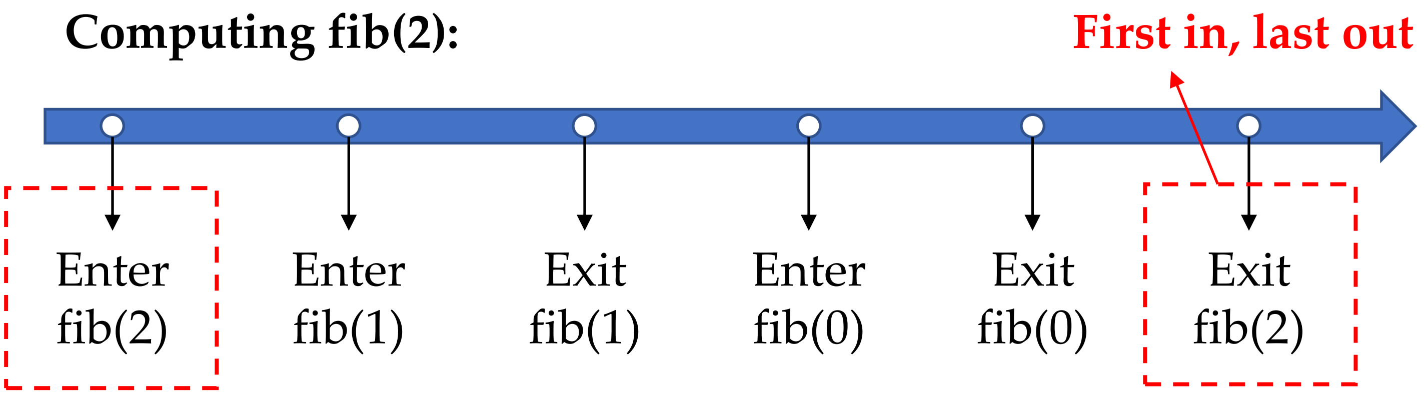
void $A()$ {
    Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

Recursive-descent parsing has recursive calls.

Can we design a non-recursive parser?

How Is Recursion Handled?

```
int fib(int n) {
    if(n <= 1) return n;
    else {
        int a = fib(n-1) + fib(n-2);
        return a;
    }
}
```



Non-Recursive Predictive Parsing

- A non-recursive predictive parser can be built by **explicitly maintaining a stack** (not implicitly via recursive calls)
  - **Input buffer** contains the string to be parsed, ending with $
  - **Stack** holds a sequence of grammar symbols with $ at the bottom. Initially, the stack contains only $ and the start symbol S on top of $

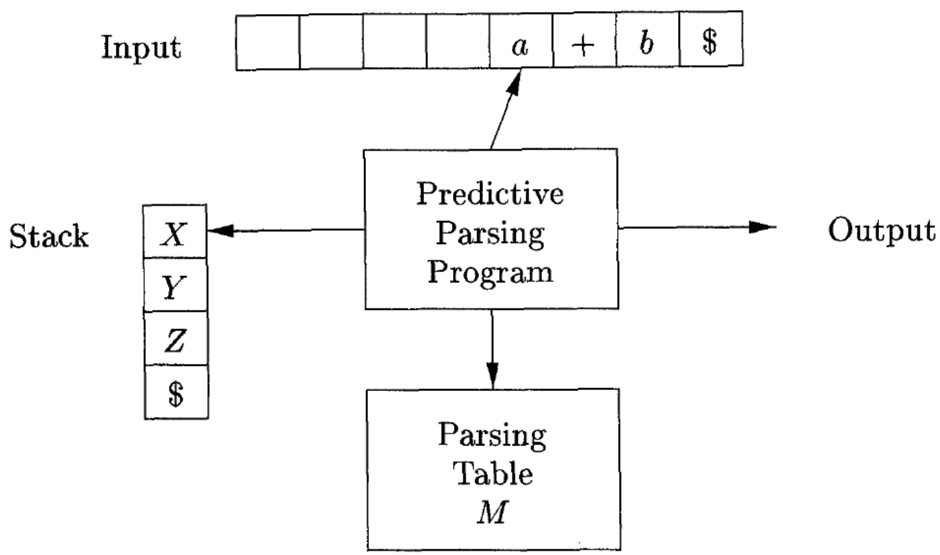
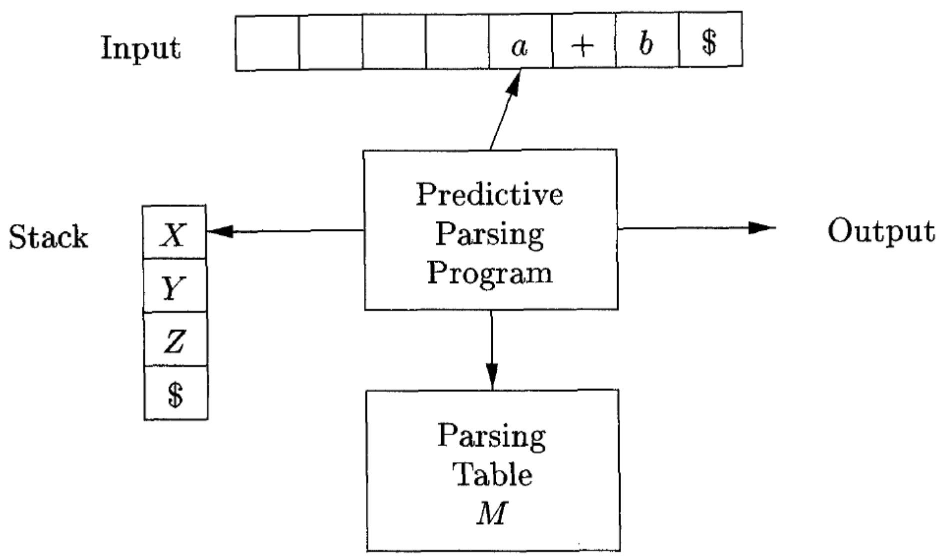


Table-Driven Predictive Parsing

- **Input:** A string ω and a parsing table M for grammar G
- **Output:** If ω is in $L(G)$, a leftmost derivation of ω (input buffer and stack are both empty); otherwise, an error indication



Initially, the input buffer contains $\omega\$$.

The start symbol S of G is on top of the stack, above $\$$.

Example

$E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid \mathbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$\underline{E}\$$	$\underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\$$	

Example

$E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid \mathbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to * FT' \mid \epsilon$ $\quad F \to (E) \mid \textbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	
	$TE'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $T \to FT'$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to * FT' \mid \epsilon$ $\quad F \to (E) \mid \textbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	
	$TE'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $T \to FT'$
	$\underline{\textbf{id}}\ T'E'\$$	$\underline{\textbf{id}} + \textbf{id} * \textbf{id}\$$	output $F \to \textbf{id}$

Example

$E \to TE' \quad E' \to +TE' \mid \epsilon$

$T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid \mathbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
id	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match **id**

Example

$E \to TE' \quad E' \to +TE' \mid \epsilon$

$T \to FT' \quad T' \to *FT' \mid \epsilon \quad F \to (E) \mid \mathbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
id	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match **id**
id	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$

Example

$E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid \textbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	
	$TE'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $T \rightarrow FT'$
	$\textbf{id}\ T'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $F \rightarrow \textbf{id}$
$\textbf{id}$	$T'E'\$$	$+ \textbf{id} * \textbf{id}\$$	match $\textbf{id}$
$\textbf{id}$	$E'\$$	$+ \textbf{id} * \textbf{id}\$$	output $T' \rightarrow \epsilon$
$\textbf{id}$	$\underline{+}\ TE'\$$	$\underline{+}\ \textbf{id} * \textbf{id}\$$	output $E' \rightarrow + TE'$

Example

$E \to TE' \quad E' \to +TE' \mid \epsilon$

$T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
...	...	...	...
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	\$	\$	output $E' \to \epsilon$

There are eight more steps in the middle before accepting.

The parser announce success when both stack and input are empty.

Example

$E \to TE' \qquad E' \to +TE' \mid \epsilon$

$T \to FT' \qquad T' \to * FT' \mid \epsilon \qquad F \to (E) \mid \mathbf{id}$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

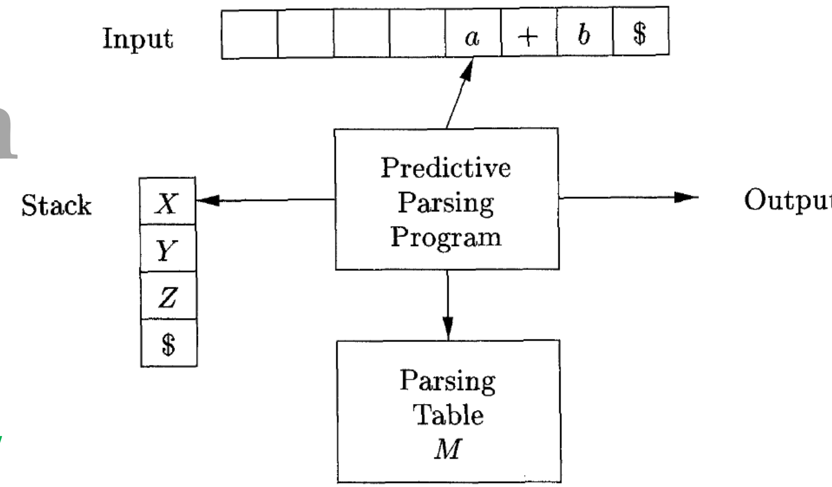
MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to +TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
...	...	...	...
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Leftmost derivation

Matched part
+
Stack content
(from top to bottom)
=
A left-sentential form

总是最左句型

Parsing Algorithm



let a be the first symbol of ω;
let X be the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if ($X = a$) pop the stack and let a be the next symbol of ω;
    else if (X is a terminal) *error*(); /* X can only match a, cannot be another terminal */
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top; /* order is critical */
    }
    let X be the top stack symbol;
}