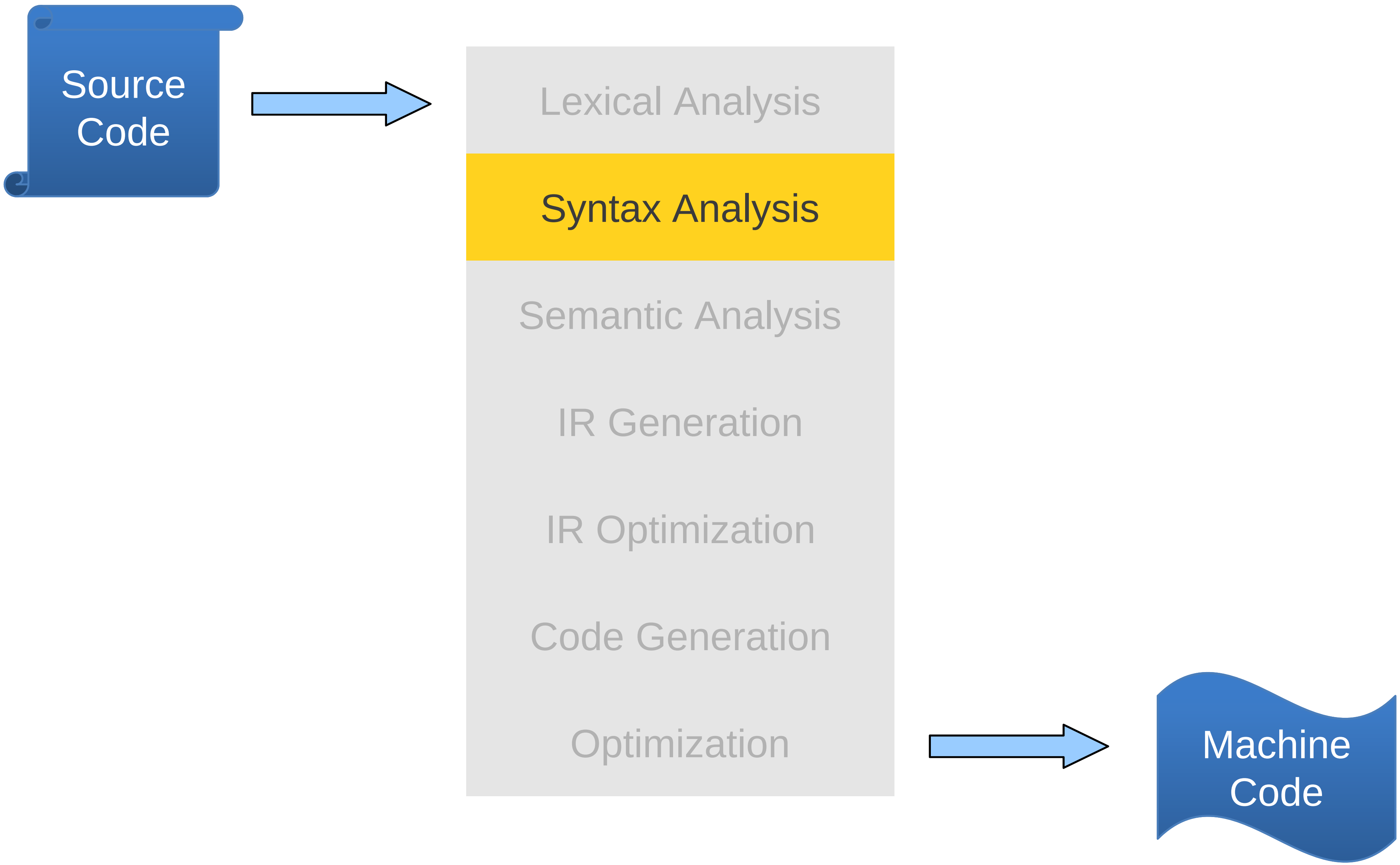
Compilers and Interpreters

Top-Down Parsing

Where are we ?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Review

- Goal of syntax analysis: recover the intended structure of the program.
- Idea: Use a context-free grammar to describe the programming language.
- Given a sequence of tokens, look for a parse tree that generates those tokens.
- Recovering this syntax tree is called parsing.

Different Types of Parsing

- Top-Down Parsing
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.

- Bottom-Up Parsing
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

Top-Down Parsing

The parse tree is created top to bottom (from root to leaves).

By always replacing the leftmost non-terminal symbol via a production rule, we are guaranteed of developing a parse tree in a left-to-right fashion that is consistent with scanning the input.

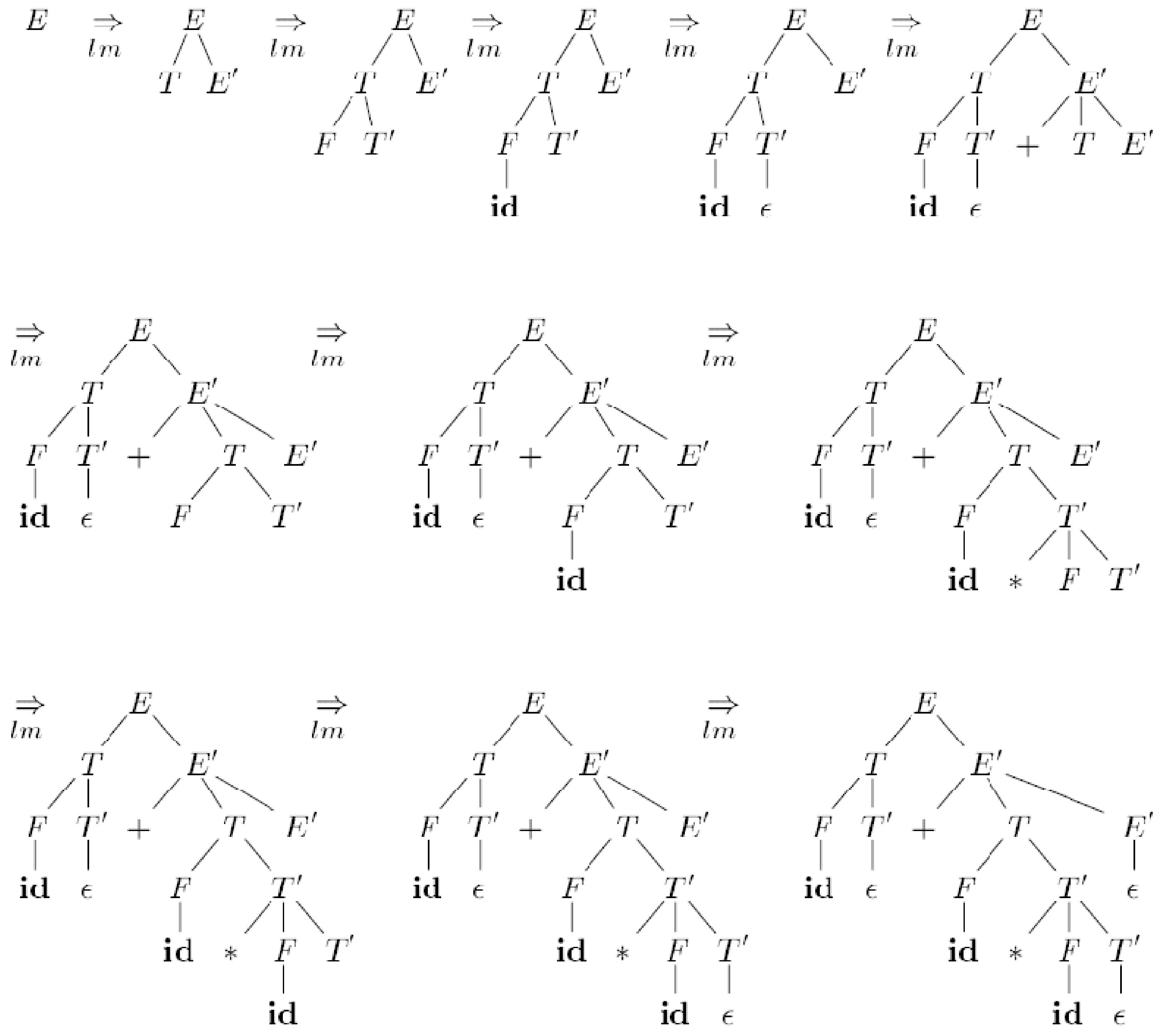


Figure 4.12: Top-down parse for $\mathbf{id} + \mathbf{id} * \mathbf{id}$

G: $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid i$

Top-Down Parsing (cont.)

Top-down parser

- **Recursive-Descent Parsing**

  Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)

  It is a general parsing technique, but not widely used.

  Not efficient

- **Predictive Parsing**

  no backtracking

  efficient

  needs a special form of grammars (**LL(1) grammars**).

  Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.

  Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

A ⇒ aBc ⇒ adDc ⇒ adec

(scan a, scan d, scan e, scan c - accept!)

Parsing Top-Down

Goal: construct a **leftmost derivation** of string while reading in sequential token stream (**left-to-right**)

S → E + S | E
E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ 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
→ ...		

Problem with Top-Down Parsing

Want to decide which production to apply based on next symbol (token).

$S \rightarrow E + S \mid E$
$E \rightarrow num \mid (S)$

Ex1: "(1)" $S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$
Ex2: "(1)+2" $S \rightarrow \underline{E+S} \rightarrow (S)+S \rightarrow (E)+S \rightarrow (1)+E \rightarrow (1)+2$

How did you know to pick E+S in Ex2, if you picked E followed by (S), you couldn't parse it?

Grammar is Problem

$S \rightarrow E + S \mid E$
$E \rightarrow num \mid (S)$

- **This grammar cannot be parsed top-down with only a single look-ahead symbol!**
- **Not LL(1) = Left-to-right scanning, Left-most derivation, 1 look-ahead symbol**
- **Is it LL(k) for some k?**
- **If yes, then can rewrite grammar to allow top-down parsing: create LL(1) grammar for same language**

Making a Grammar LL(1)

S → E + S
S → E
E → num
E → (S)

S → ES'
S' → ε
S' → +S
E → num
E → (S)

- Problem: Can't decide which S production to apply until we see the symbol after the first expression

- Left-factoring: **Factor** common S prefix, add new non-terminal S' at decision point. S' derives (+S)*

- Also: **Convert left recursion** to right recursion

Left Factoring

- Needed to produce grammar suitable for predictive top-down parsing

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

After left factoring becomes:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

$S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

After left factoring becomes:

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b$

Left Factoring

- Algorithm: Left Factoring a grammer
- INPUT: Grammar G
- OUTPUT: An equivalent left-factored grammar

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. $\square$

Parsing with New Grammar

$S \rightarrow ES'$ $\quad$ $S' \rightarrow \varepsilon \mid +S$ $\quad$ $E \rightarrow \text{num} \mid (S)$

Partly-derived String	**Lookahead**	**parsed part unparsed part**
→ES’	(	(1+2+(3+4))+5
→(S)S’	1	(1+2+(3+4))+5
→(ES’)S’	1	(1+2+(3+4))+5
→(1S’)S’	+	(1+2+(3+4))+5
→(1+ES’)S’	2	(1+2+(3+4))+5
→(1+2S’)S’	+	(1+2+(3+4))+5
→(1+2+S)S’	(	(1+2+(3+4))+5
→(1+2+ES’)S’	(	(1+2+(3+4))+5
→(1+2+(S)S’)S’	3	(1+2+(3+4))+5
→(1+2+(ES’)S’)S’	3	(1+2+(3+4))+5
→(1+2+(3S’)S’)S’	+	(1+2+(3+4))+5
→(1+2+(3+E)S’)S’	4	(1+2+(3+4))+5
→ ...		

Grammars

- **Have been using grammar for language "sums with parentheses" (1+2+(3+4))+5**
- **Started with simple, right-associative grammar**

  $S \rightarrow E + S \mid E$

  $E \rightarrow num \mid (S)$

- **Transformed it by left factoring:**

  $S \rightarrow ES'$

  $S' \rightarrow \varepsilon \mid +S$

  $E \rightarrow num\ (S)$

- **What if we start with a left-associative grammar?**

  $S \rightarrow S + E \mid E$

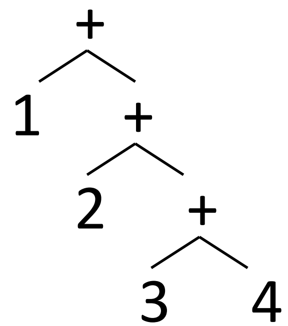
  $E \rightarrow num \mid (S)$

Reminder: Left vs Right Associativity

Consider a simpler string on a simpler grammar: “1 + 2 + 3 + 4”

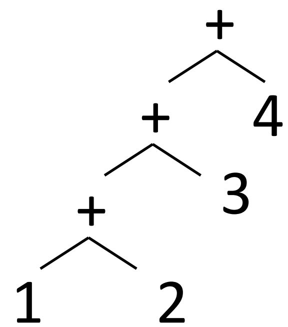
Right recursion : right associative

S → E + S
S → E
E → num



Left recursion : left associative

S → S + E
S → E
E → num



Left Recursion

S → S + E
S → E
E → num

“1 + 2 + 3 + 4”

derived string	lookahead	read/unread
S	1	1+2+3+4
S+E	1	1+2+3+4
S+E+E	1	1+2+3+4
S+E+E+E	1	1+2+3+4
E+E+E+E	1	1+2+3+4
1+E+E+E	2	1+2+3+4
1+2+E+E	3	1+2+3+4
1+2+3+E	4	1+2+3+4
1+2+3+4	$	1+2+3+4

Is this right? If not, what's the problem?

Left-Recursive Grammars

- **Left-recursive grammars don’t work with top-down parsers: we don’t know when to stop the recursion**
- **Left-recursive grammars are NOT LL(1)!**

$$S \rightarrow S\alpha$$

$$S \rightarrow \beta$$

Eliminate Left Recursion

Consider following grammar

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

Replace by non-left-recursive productions

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

Eliminate Left Recursion (cont)

- **Replace**

  $X \rightarrow X\alpha_1 \mid \ldots \mid X\alpha_m$

  $X \rightarrow \beta_1 \mid \ldots \mid \beta_n$

- **With**

  $X \rightarrow \beta_1 X' \mid \ldots \mid \beta_n X'$

  $X' \rightarrow \alpha_1 X' \mid \ldots \mid \alpha_m X' \mid \varepsilon$

- ***See complete algorithm in Dragon book***

Eliminate Left Recursion (cont)

$$E \to E+T|T$$
$$T \to T*F|F$$
$$F \to (E)|\ i$$

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

Predictive Parsing

- **LL(1) grammar:**
  - –For a given non-terminal, the lookahead symbol **uniquely** determines the production to apply
  - –Top-down parsing = predictive parsing
  - –Driven by predictive **parsing table**

Parsing with Table

S → ES' S' → ε | +S E → num | (S)

Partly-derived String	**Lookahead**	**parsed part unparsed part**
→ES'	(	(1+2+(3+4))+5
→(S)S'	1	(1+2+(3+4))+5
→(ES')S'	1	(1+2+(3+4))+5
→(1S')S'	+	(1+2+(3+4))+5
→(1+S)S'	2	(1+2+(3+4))+5
→(1+ES')S'	2	(1+2+(3+4))+5
→(1+2S')S'	+	(1+2+(3+4))+5

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

How to Implement This?

- Table can be converted easily into a **recursive descent parser**

- 3 procedures: parse_S(), parse_S'(), and parse_E()

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser

lookahead token

```
void parse_S() {
   switch (token) {
         case num: parse_E(); parse_S'(); return;
         case '(': parse_E(); parse_S'(); return;
         default: ParseError();
   }
}
```

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser (2)

```
void parse_S’() {
   switch (token) {
         case ‘+’: token = input.read(); parse_S(); return;
         case ‘)‘: return;
         case EOF: return;
         default: ParseError();
   }
}
```

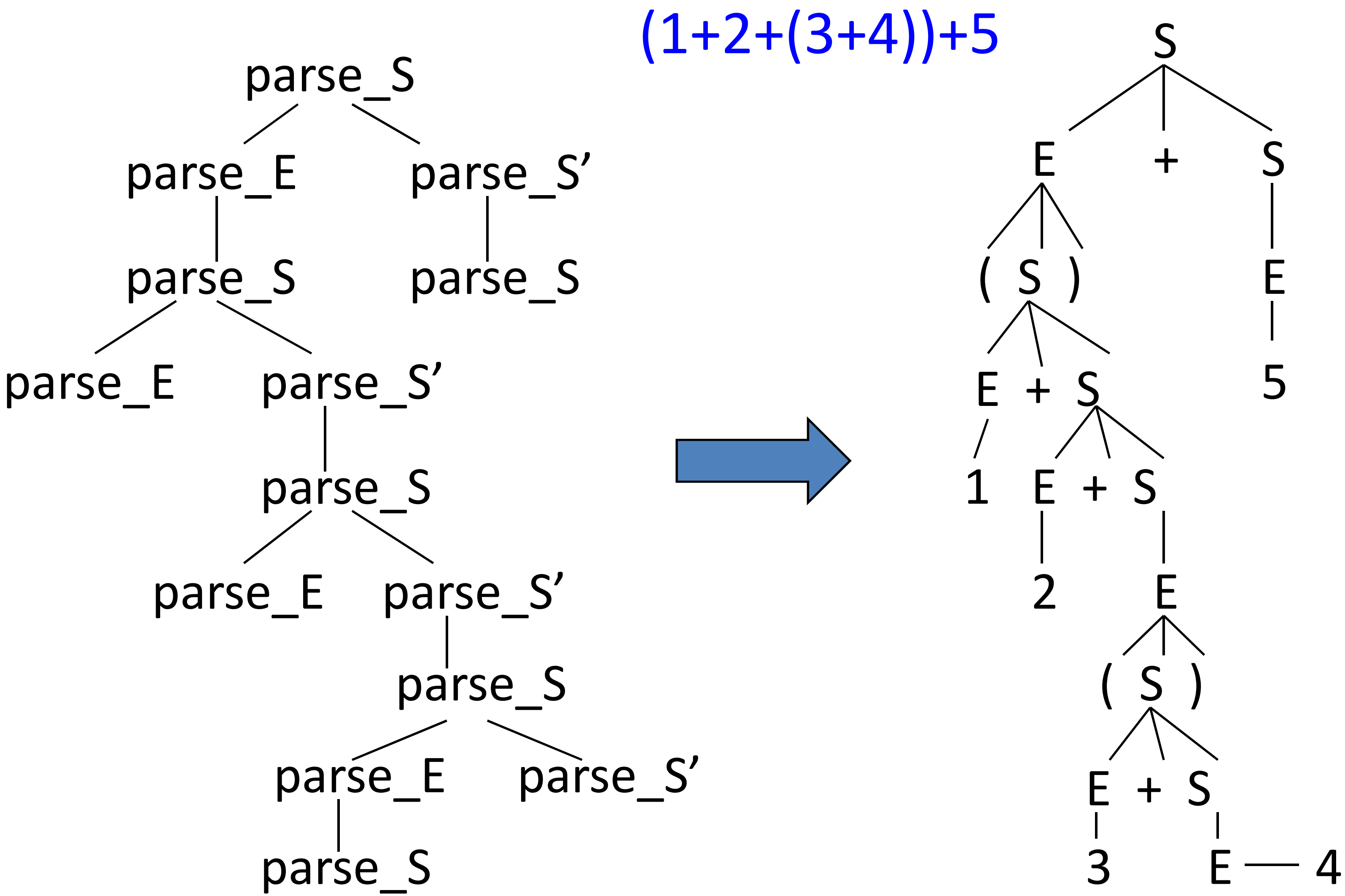
	num	+	(	)	$
S	→ ES’		→ ES’		
S’		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser (3)

```
void parse_E() {
   switch (token) {
         case number: token = input.read(); return;
         case '(': token = input.read(); parse_S();
                  if (token != ')') ParseError();
                  token = input.read(); return;
         default: ParseError();
   }
}
```

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Call Tree = Parse Tree



How to Construct Parsing Tables?

Needed: Algorithm for automatically generating a predictive parse table from a grammar

S → ES'
S' → ε | +S
E → number | (S)

??

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Building the parse table

- **Define two functions on the symbols of the grammar: FIRST and FOLLOW.**
- **For a non-terminal N, FIRST (N) is the set of terminal symbols that can start any derivation from N.**
  - First (If_Statement) = {***if***}
  - First (Expr) = {**id**, (}
- **FOLLOW (N) is the set of terminals that can appear after a string derived from N:**
  - Follow (Expr) = {+,), $ }

FIRST Set

- FIRST(**A**) = { **t** | **A** $\Rightarrow^*$ **t**$\boldsymbol{\omega}$ for some $\boldsymbol{\omega}$ }

1. If X is a terminal, FIRST(X) = {X}
2. If $X \rightarrow \varepsilon$ is a production rule, add ε to FIRST (X)
3. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production rule

Place FIRST(Y_1) in FIRST (X)

if $Y_1 \Rightarrow^* \varepsilon$, Place FIRST ($Y_2$) in FIRST(X)

if $Y_2 \Rightarrow^* \varepsilon$, Place FIRST($Y_3$) in FIRST(X)

…

if $Y_{k-1} \Rightarrow^* \varepsilon$, Place FIRST($Y_k$) in FIRST(X)

Repeat above steps until no more elements are added to any FIRST( ) set.

Checking "$Y_i \Rightarrow \varepsilon$?" essentially amounts to checking whether ε belongs to FIRST(Y_i)

Computing FIRST(X) : All Grammar Symbols - continued

Informally, suppose we want to compute

$$
\begin{aligned}
\text{FIRST}(X_1\, X_2 \ldots X_n) = {} & \text{FIRST}(X_1) \\
& + \text{FIRST}(X_2) \text{ if } \varepsilon \text{ is in FIRST}(X_1) \\
& + \text{FIRST}(X_3) \text{ if } \varepsilon \text{ is in FIRST}(X_2) \\
& \ldots \\
& + \text{FIRST}(X_n) \text{ if } \varepsilon \text{ is in FIRST}(X_{n-1})
\end{aligned}
$$

Note 1: Only add ε to FIRST ($X_1\, X_2 \ldots X_n$) if ε is in FIRST (X_i) for all i

Note 2: For FIRST(X_1), if $X_1 \rightarrow Z_1\, Z_2 \ldots Z_m$, then we need to compute FIRST($Z_1\, Z_2 \ldots Z_m$) !

Motivation Behind FIRST

- **Is used to help find the appropriate reduction to follow given the top-of-the-stack non-terminal and the current input symbol.**
- **If $A \rightarrow \alpha$, and a is in FIRST(α), then when *a*=input, replace A with α. (*a* is one of first symbols of α, so when A is on the stack and a is input, POP A and PUSH α.)**

Example:

$A \rightarrow aB \mid bC$

$B \rightarrow b \mid dD$

$C \rightarrow c$

$D \rightarrow d$

FIRST Example

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

FIRST(TE') = {(,id}
FIRST(+TE') = {+}
FIRST(ε) = {ε}
FIRST(FT') = {(,id}
FIRST(*FT') = {*}
FIRST(ε) = {ε}
FIRST((E)) = {(}
FIRST(id) = {id}

FIRST(F) = {(,id}
FIRST(T') = {*, ε}
FIRST(T) = {(,id}
FIRST(E') = {+, ε}
FIRST(E) = {(,id}

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

input:
i+i $

E: FIRST(TE′) = { (, i }
E′: FIRST(+TE′) = { + }   FIRST(ε) = { ε }
T: FIRST(FT′) = { (, i }
T′: FIRST(*FT′) = { * }   FIRST(ε) = {ε}
F: FIRST((E)) = { (}   FIRST(i) = {i}

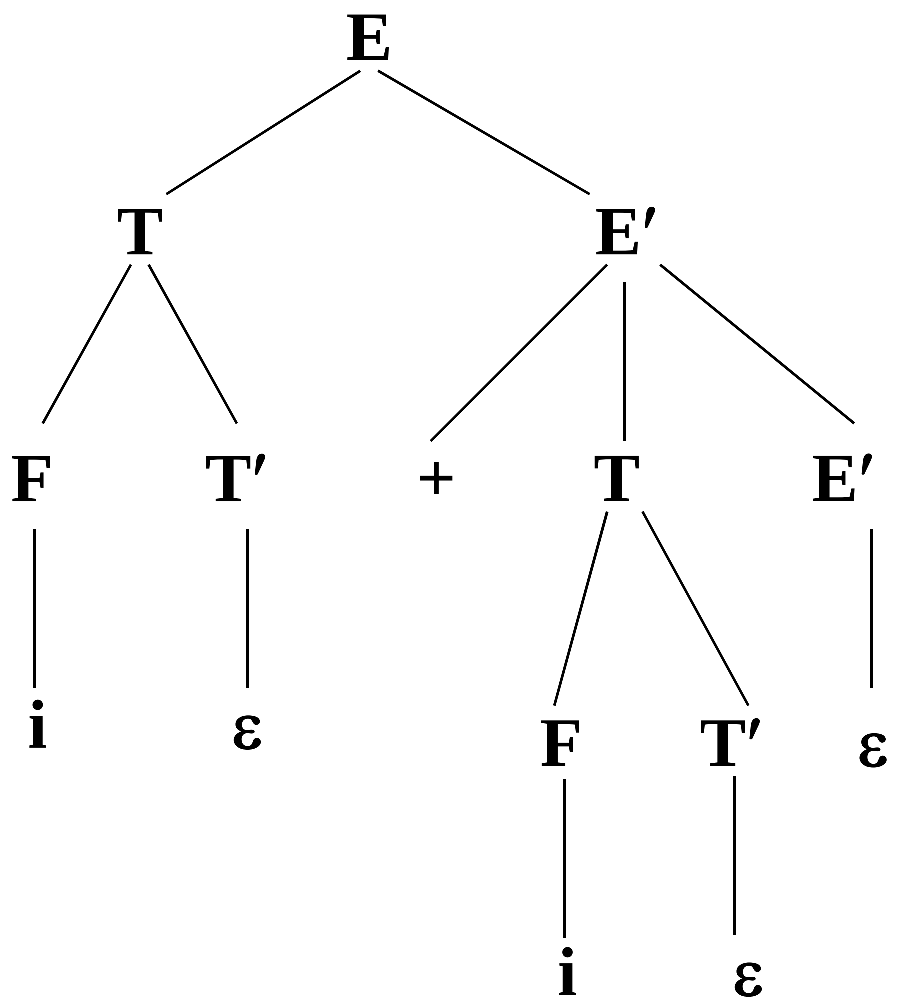
i + i   i∈FIRST(TE′)

i + i   i∈FIRST(FT ′)

i + i   i∈FIRST(i)

i + i   use T′→ε

……



Left Most Derivation of the Example

$$
\begin{aligned}
E &\Rightarrow TE' \\
&\Rightarrow FT'E' \\
&\Rightarrow iT'E' \\
&\Rightarrow i\varepsilon E' \\
&\Rightarrow i\varepsilon + TE' \\
&\Rightarrow i\varepsilon + FT'E' \\
&\Rightarrow i\varepsilon + iT'E' \\
&\Rightarrow i\varepsilon + i\varepsilon E' \\
&\Rightarrow i\varepsilon + i\varepsilon\varepsilon = i+i
\end{aligned}
$$

- $E \rightarrow TE'$

$T \rightarrow FT'$

$F \rightarrow i$

$T' \rightarrow \varepsilon$

$\boxed{E' \rightarrow +TE'}$

$T \rightarrow FT'$

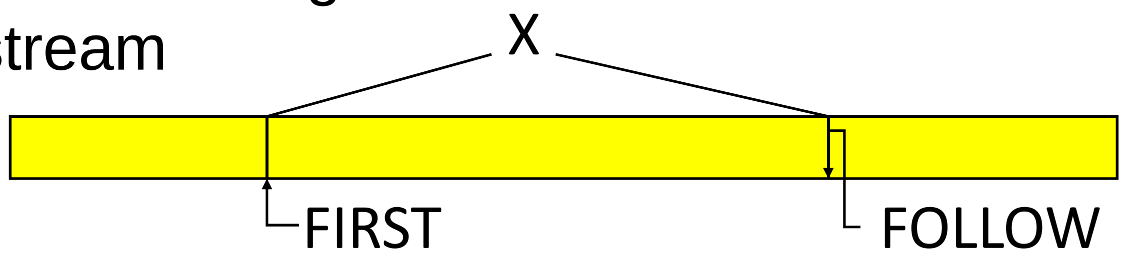
$F \rightarrow i$

$T' \rightarrow \varepsilon$

$E' \rightarrow \varepsilon$

Constructing Parse Tables

- **Can construct predictive parser if:**
  - –For every non-terminal, every lookahead symbol can be handled by at most 1 production
- **FIRST(β) for an arbitrary string of terminals and non-terminals β is:**
  - –Set of symbols that might begin the fully expanded version of β
- **FOLLOW(X) for a non-terminal X is:**
  - –Set of symbols that might follow the derivation of X in the input stream



FOLLOW Set

FOLLOW: Let A be a non-terminal. FOLLOW(A) is the set of terminals a that can appear directly to the right of A in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).

NOTE: If $S \Rightarrow \alpha A$, then \$ is FOLLOW(A).

FOLLOW Set (cont.)

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

- Place \$ in FOLLOW(S) , where S is the start symbol, and \$ is the input right endmarker.
- If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B) .
- If there is a production $A \rightarrow B\beta$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ε, then everything in FOLLOW(A) is in FOLLOW(B) .

FOLLOW Example

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

FIRST(F) = {(,id}
FIRST(T') = {*, ε}
FIRST(T) = {(,id}
FIRST(E') = {+, ε}
FIRST(E) = {(,id}

FOLLOW(E) = {), $ }
FOLLOW(E') = {), $ }
FOLLOW(T) = { +,), $ }
FOLLOW(T') = { +,), $ }
FOLLOW(F) = {*, +,), $ }

Motivation Behind FOLLOW

- **Is used when FIRST has a conflict, to resolve choices, or when FIRST gives no suggestion. When $\alpha \to \in$ or $\alpha \Rightarrow^* \varepsilon$, then what follows A dictates the next choice to be made.**
- **If A $\to \alpha$, and *b* is in FOLLOW(A), then when $\alpha \Rightarrow^* \varepsilon$ and *b* is an input character, then we expand A with α , which will eventually expand to ε, of which *b* follows! ($\alpha \Rightarrow^* \varepsilon$: i.e., FIRST(α) contains ε.)**

Parse Table Entries

- **Consider a production X → β**
- **Add → β to the X row for each symbol in FIRST(β)**
- **If β can derive ε (β is nullable), add → β for each symbol in FOLLOW(X)**
- **Grammar is LL(1) if no conflicting entries**

S → ES'
S' → ε | +S
E → number | (S)

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Computing Nullable

- **X is nullable if it can derive the empty string:**
  - –If it derives ε directly (X → ε)
  - –If it has a production X → YZ ... where all RHS symbols (Y,Z) are nullable
- **Algorithm: assume all non-terminals are non-nullable, apply rules repeatedly until no change**

S → ES'
S' → ε | +S
E → number | (S)

Only S' is nullable

Compute FIRST

$S \rightarrow ES'$

$S' \rightarrow \varepsilon \mid +S$

$E \rightarrow \text{number} \mid (S)$

FIRST(S) = FIRST(E) = {num, (}

FIRST(S') = {ε, + }

FIRST(E) = {num, (}

Compute FOLLOW

S → ES'
S' → ε | +S
E → number | (S)

FIRST(S) = {num, (}
FIRST(S') = {ε, + }
FIRST(E) = { num, (}

Step 1: FOLLOW(S) = {$}
Step 2: S → ES'   FOLLOW(E) += {FIRST(S') - ε} = {+}
S' → ε | +S
E → num | (S)   FOLLOW(S) += {FIRST(')') - ε} = {$,) }
Step 3: S → ES'   FOLLOW(E) += FOLLOW(S) = {+,$,)}
(because S' is nullable)
FOLLOW(S') += FOLLOW(S) = {$,)}

Parse Table

FIRST(S) = {num, (}
FIRST(S') = {ε, + }
FIRST(E) = { num, (}

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

- Consider a production X → β
- Add → β to the X row for each symbol in FIRST(β)
- If β can derive ε (β is nullable), add → β for each symbol in FOLLOW(X)

S → ES'
S' → ε | +S
E → number | (S)

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Top-Down Parsing Up to This Point

- **Now we know**
  - How to build parsing table for an LL(1) grammar (ie FIRST/FOLLOW)
  - How to construct recursive-descent parser from parsing table
  - Call tree = parse tree

Predictive Parser

a grammar → eliminate left recursion → left factor → a grammar suitable for pedictive parsing (a LL(1) grammar) no %100 guarantee.

When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the **current symbol** in the input string.

$$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$$

input: ... a ....

current token

Predictive Parser

stmt $\rightarrow$ **if** expr **then** stmt **else** stmt
|**while** expr **do** stmt
|**begin** stmt_list **end**
|**for** stmt…

- When we are trying to write the non-terminal ***stmt***, if the current token is **if** we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We **eliminate the left recursion** in the grammar, and **left factor** it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive-Descent Parsing

```
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 */;
    }
}
```

A typical procedure for a nonterminal in a top-down parse

Recursive Predictive Parsing

$A \rightarrow aBb \mid bAB$

```
proc A {
   case of the current token {
         'a':   - match the current token with a, and move to the
                  next token;
                - call 'B';
                - match the current token with b, and move to the
                  next token;
         'b':   - match the current token with b, and move to the
                  next token;
                - call 'A';
                - call 'B';
   }
}
```

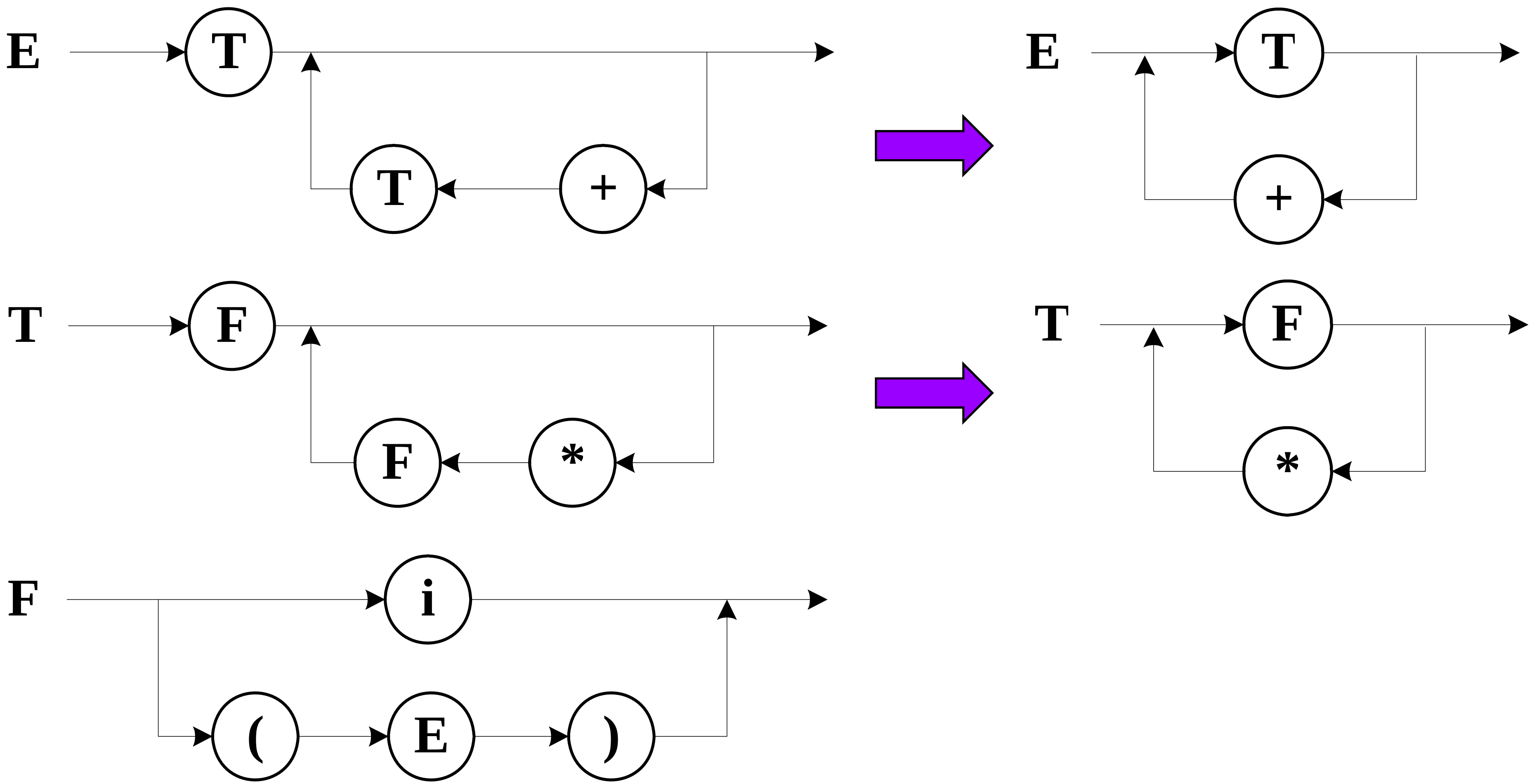
F → (E) | number

```
PROCEDURE F;
BEGIN
  IF token = '(' THEN
  BEGIN
     match('(');
    E;
    match(')');
  END;
  ELSE
     IF token = number
    THEN
         match(number);
     ELSE ERROR;
END
```

```
PROCEDURE match(expectedToken);
BEGIN
  IF token = expectedToken THEN
  BEGIN
       getNextToken();
  END
  ELSE ERROR;
END
```

- G: $E \rightarrow T \mid E+T$
  $T \rightarrow F \mid T*E$
  $F \rightarrow i \mid (E)$

- EBNF expression:
  G: $E \rightarrow T\{+T\}$
  $T \rightarrow F\{*E\}$
  $F \rightarrow i \mid (E)$



```
PROCEDURE MAIN;
    BEGIN
        token = nexttoken();
        E
    END

PROCEDURE match(t:token);
    BEGIN
        IF token = t THEN
            getNextToken()
        ELSE   ERROR
END
```

$E \rightarrow T\{+T\}$

```
PROCEDURE E;
  BEGIN
     T;
     WHILE token = '+' DO
        BEGIN
           match('+');
           T;
        END
  END
```

$F \rightarrow i \mid (E)$

```
PROCEDURE F;
  BEGIN
    IF token = i THEN match(i)
     ELSE
      IF token =‘(’ THEN
         BEGIN
           match(‘(’);
           E;
           IF token = ‘)’ THEN  match(‘)’)
           ELSE ERROR;
         END
      ELSE  ERROR
  END
```

$T \rightarrow F\{*F\}$

```
PROCEDURE T;
  BEGIN
    F;
    WHILE token = ‘*’ DO
      BEGIN
        match(‘*’);
        F;
      END
  END
```

Simple Predictive Parser: LL(1)

- Top-down, predictive parsing:
  - L: Left-to-right scan of the tokens
  - L: Leftmost derivation.
  - (1): One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
- When expanding a nonterminal, we predict the production to use by looking at the next token of the input. The decision is forced.

LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
  - Both α and β cannot derive strings starting with same terminals.

    $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$, $\quad FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset \quad (1 \leq i \neq j \leq n)$
  - At most one of α and β can derive to ε.
  - If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

    If $\varepsilon \in FIRST(\alpha_j)(1 \leq i \leq n)$, $\quad$ then $FIRST(\alpha_i) \cap FOLLOW(A) = \emptyset$

NOW predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.

Constructing LL(1) Parsing Table

Algorithm 4.31
INPUT: Grammar G.
OUTPUT: Parsing table M.

METHOD : For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal *a* in FIRST(*A*) , add $A \rightarrow \alpha$ to M[*A*, *a*] .
2. If E is in FIRST(*a*) , then for each terminal b in FOLLOW(*A*) , add $A \rightarrow \alpha$ to M [*A*, *b*] . If E is in FIRsT(*a*) and $ is in FOLLOW(*A*) , add $A \rightarrow \alpha$ to M[*A*, $] as well.
3. If, after performing the above, there is no production at all in M[*A*, *a*] , then set M[*A*, *a*] to error (which we normally represent by an empty entry in the table) .

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

- FIRST Set

FIRST(TE′) = { (, i }

FIRST(+TE′) = { + } FIRST(ε) = { ε }

FIRST(FT′) = { (, i }

FIRST(*FT′) = { * } FIRST(ε) = {ε}

FIRST((E)) = { (} FIRST(i) = {i}

- FOLLOW Set

FOLLOW(E) = {), $ }

FOLLOW(E′) = {), $ }

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

FOLLOW(T′) = {+,), $ }

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

V_N	input symbol					
	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	$F \to i$			$F \to (E)$		

LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol **$**.

output

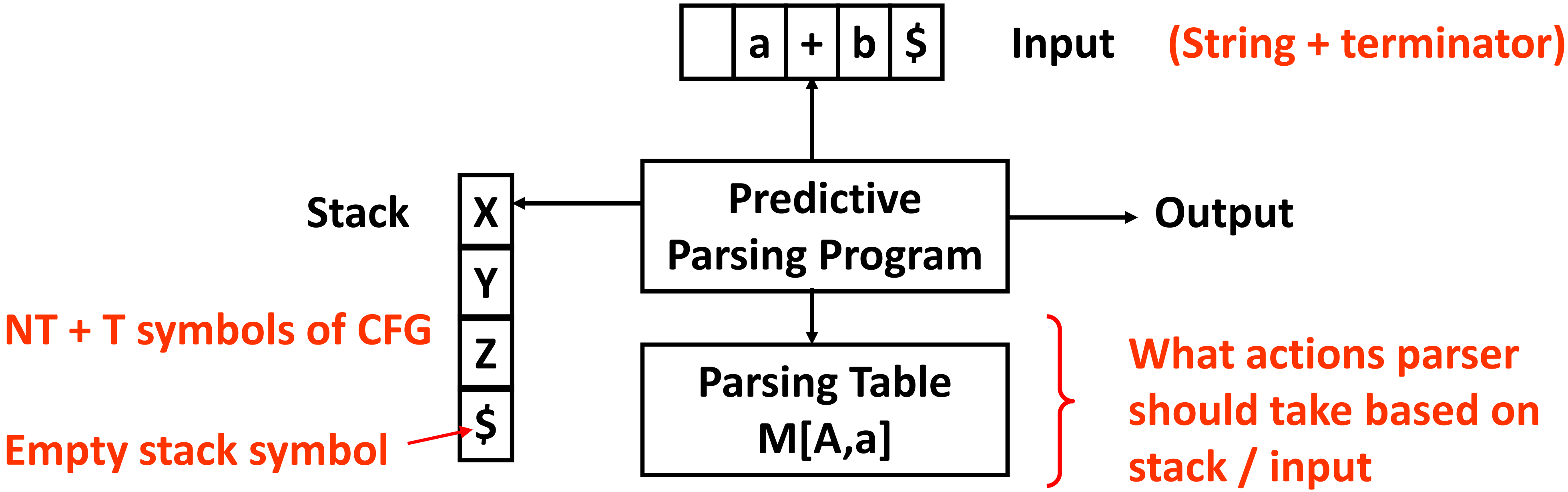
- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol **$**.
- initially the stack contains only the symbol **$** and the starting symbol **S**.
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

Non-Recursive / Table Driven



General parser behavior: (X : top of stack *a* : current input)

1. When X=*a* = $ halt, accept, success
2. When X=*a* ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X,a]
   if it is an error, then call recovery routine
   if M[X,a] = {X → UVW}, POP X, PUSH W,V,U
   DO NOT expend any input

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

input: $i_1 * i_2 + i_3$

stack	input	output	stack	input	output
$\$E$	$i_1*i_2+i_3\$$		$\$E'$	$+i_3\$$	$T' \to \varepsilon$
$\$E'T$	$i_1*i_2+i_3\$$	$E \to TE'$	$\$E'T+$	$+i_3\$$	$E' \to +TE'$
$\$E'T'F$	$i_1*i_2+i_3\$$	$T \to FT'$	$\$E'T$	$i_3\$$	
$\$E'T'i$	$i_1*i_2+i_3\$$	$F \to i$	$\$E'T'F$	$i_3\$$	$T \to FT'$
$\$E'T'$	$*i_2+i_3\$$		$\$E'T'i$	$i_3\$$	$F \to i$
$\$E'T'F*$	$*i_2+i_3\$$	$T' \to *FT'$	$\$E'T'$	$\$$	
$\$E'T'F$	$i_2+i_3\$$		$\$E'$	$\$$	$T' \to \varepsilon$
$\$E'T'i$	$i_2+i_3\$$	$F \to i$	$\$$	$\$$	$E' \to \varepsilon$
$\$E'T'$	$+i_3\$$				accept

$E \Rightarrow \underline{T}E' \Rightarrow \underline{F}T'E' \Rightarrow i\underline{T'}E' \Rightarrow i*\underline{F}T'E' \Rightarrow i*i\underline{T'}E' \Rightarrow \ldots \Rightarrow i*i+i$

Revisit LL(1) Grammar

LL(1) grammars

== there have no multiply-defined entries in the parsing table.

Properties of LL(1) grammars:

- Grammar can't be ambiguous or left recursive
- Grammar is LL(1) $\Leftrightarrow$ when $A \rightarrow \alpha \mid \beta$

1. α and β do not derive strings starting with the same terminal a

2. Either α or β can derive ε, but not both.

Note: It may not be possible for a grammar to be manipulated into an LL(1) grammar

A Grammar which is not LL(1)

- A left recursive grammar cannot be a LL(1) grammar.
  - $A \to A\alpha \mid \beta$
    - any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \to \alpha\beta_1 \mid \alpha\beta_2$
    - any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.

$S \to iEtSS'|a \quad S' \to eS|\varepsilon \quad E \to b$

FIRST(S) = {i,a} FIRST(iEtSS′) = {i} FIRST(a)={a}

FIRST(S′) = {e, ε} FIRST(eS) = {e} FIRST(ε)={ε}

FIRST(E) = {b} FIRST(b) = {b}

FELLOW(S) = {e, $}

FELLOW(S′) = {e, $}

FELLOW(E) = {t}

V_N	input symbol					
	a	b	e	i	T	$
S	$S \to a$			$S \to iEtSS'$		
S′			$S' \to eS$ ~~$S' \to \varepsilon$~~			$S' \to \varepsilon$
E		$E \to b$				

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Global-Correction
  - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Error Recovery Techniques (cont.)

- Error-Productions (used in GCC etc.)
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  – All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.

Panic-Mode Error Recovery in LL(1) Parsing (cont.)

- A simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery - Example

$S \to AbS \mid e \mid \varepsilon$
$A \to a \mid cAd$

FOLLOW(S)={$}
FOLLOW(A)={b,d}

	a	b	c	d	e	$
S	$S \to AbS$	*sync*	$S \to AbS$	*sync*	$S \to e$	$S \to \varepsilon$
A	$A \to a$	*sync*	$A \to cAd$	*sync*	*sync*	*sync*

stack	input	output
$S	aab$	$S \to AbS$
$SbA	aab$	$A \to a$
$Sba	aab$	
$Sb	ab$	
	Error: missing b, inserted	
$S	ab$	$S \to AbS$
$SbA	ab$	$A \to a$
$Sba	ab$	
$Sb	b$	
$S	$	$S \to \varepsilon$
$	$	accept

Panic-Mode Error Recovery – Example

$S \to AbS \mid e \mid \varepsilon$
$A \to a \mid cAda$

	a	b	c	d	e	$
S	$S \to AbS$	***sync***	$S \to AbS$	***sync***	$S \to e$	$S \to \varepsilon$
A	$A \to a$	***sync***	$A \to cAd$	***sync***	***sync***	***sync***

stack	**input**	**output**
$S	ceadb$	$S \to AbS$
$SbA	ceadb$	$A \to cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	Error:unexpected e (illegal A)
	(Remove all input tokens until first b or d, pop A)	
$Sbd	db$	
$Sb	b$	
$S	$	$S \to \varepsilon$
$	$	accept

G(E): $E \rightarrow TE'$ $E' \rightarrow +TE' \mid \varepsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid i$

FOLLOW(E) = {), \$ }

FOLLOW(E′) = {), \$ }

FOLLOW(T) = { +,), \$ }

FOLLOW(T′) = {+,), \$ }

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

V_N	input symbol					
	i	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E′		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T′		$T' \rightarrow \varepsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow i$	synch	synch	$F \rightarrow (E)$	synch	synch

V_N	input symbol					
	i	+	*	(	)	$
E	E→TE′			E→TE′	synch	synch
E′		E′→+TE′			E′→ε	E′→ε
T	T→FT′	synch		T→FT′	synch	synch
T′		T′→ε	T′→* FT′		T′→ε	T′→ε
F	F→i	synch	synch	F→(E)	synch	synch

Stack	Input	Actions
$E	)x*+y$	Skip ‘)’
$E	x*+y$	
$E’T	x*+y$	E→TE′
$E’T’F	x*+y$	T→F
$E’T’x	x*+y$	F→x
$E’T’	*+y$	
$E’T’F*	*+y$	T′→* FT′
$ E’T’F	+y$	

V_N	input symbol					
	i	+	*	(	)	$
E	E→TE′			E→TE′	synch	synch
E′		E′→+TE′			E′→ε	E′→ε
T	T→FT′	synch		T→FT′	synch	synch
T′		T′→ε	T′→* FT′		T′→ε	T′→ε
F	F→i	synch	synch	F→(E)	synch	synch

Stack	Input	Actions
$ E'T'	+y$	
$E'T'	+y$	T′→ε
$E'	+y$	
$E'T+	+y$	E′→+TE′
$E'T	y$	
$E'T'F	y$	T′→ε
$E'T'	$	E′→ε
$E'	$	

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

Summary

- **Top-down parsing** tries to derive the user's program from the start symbol.
- **Leftmost BFS** is one approach to top-down parsing; it is mostly of theoretical interest.
- **Leftmost DFS** is another approach to top-down parsing that is uncommon in practice.
- **LL(1)** parsing scans from left-to-right, using one token of lookahead to find a leftmost derivation.
- **FIRST sets** contain terminals that may be the first symbol of a production.
- **FOLLOW sets** contain terminals that may follow a nonterminal in a production.
- **Left recursion** and **left factorability** cause LL(1) to fail and can be mechanically eliminated in some cases.

Summary (cont.)

- Top Down parsing
  1. Rewrite grammars if necessary
  2. onstruct LL(1) predictive tables
  3. predictive parsing using the predictive table

- We’ve identified its shortcomings:

  Not all grammars can be made LL(1) !

Next Time

- Top-Down Parsing
  - Recursive descent parsing
  - Predictive parsing
  - LL(1)
- Bottom-Up Parsing
  - Shift-Reduce Parsing
  - LR parser

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
  - Chapter 4.1, 4.2, 4.3, 4.4
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu