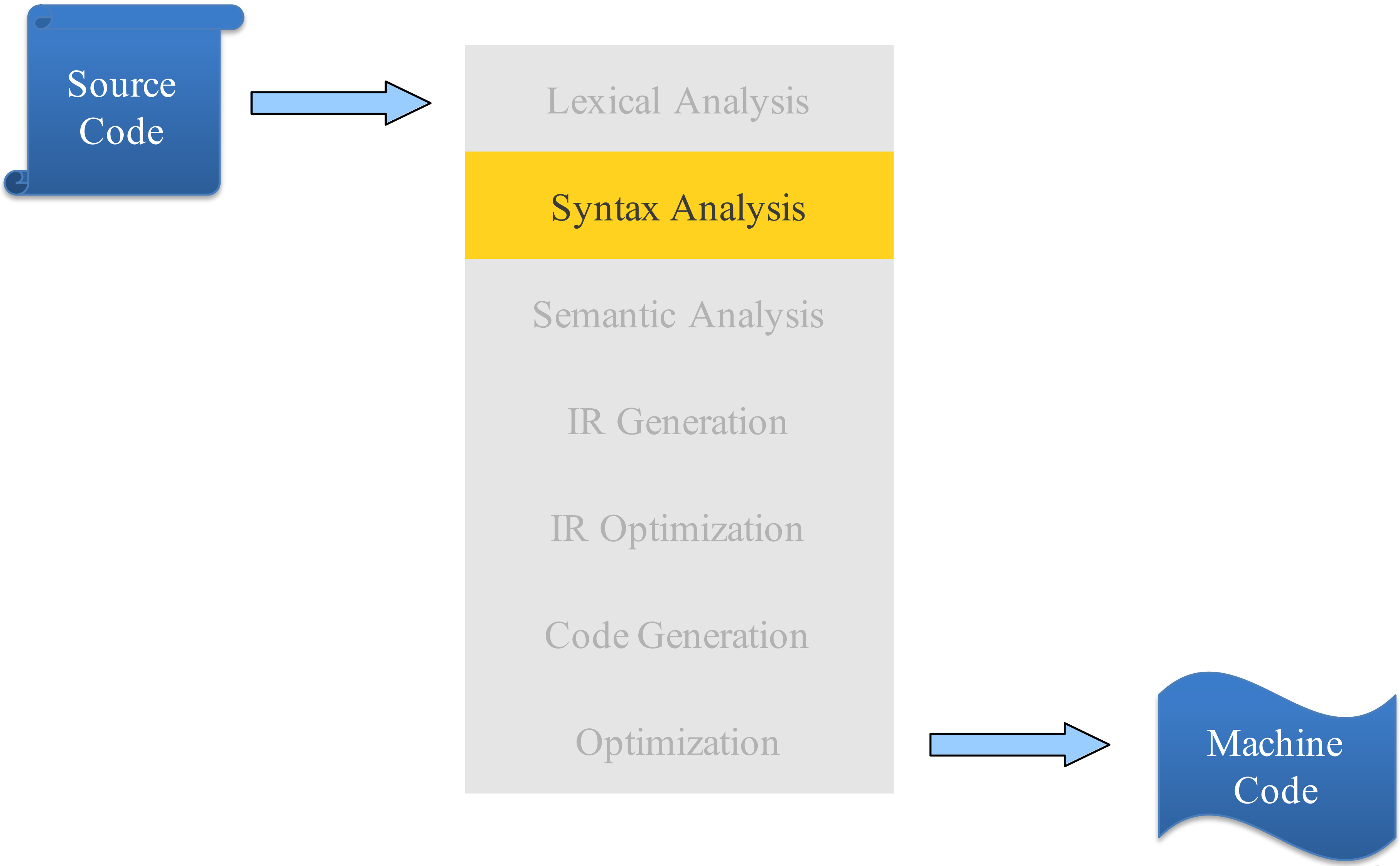
Compilers and Interpreters

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.

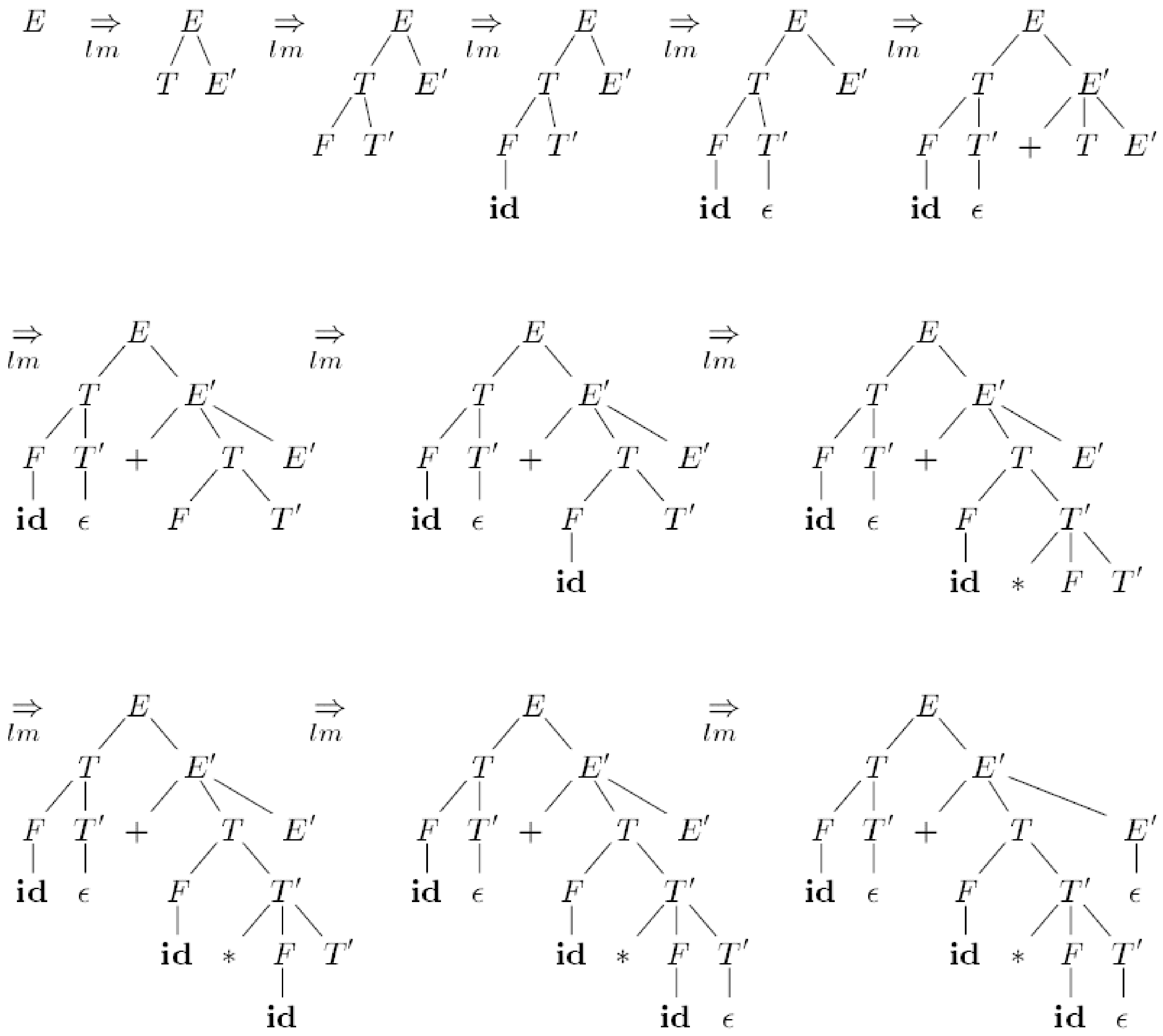


Figure 4.12: Top-down parse for **id** + **id** $*$ **id**

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

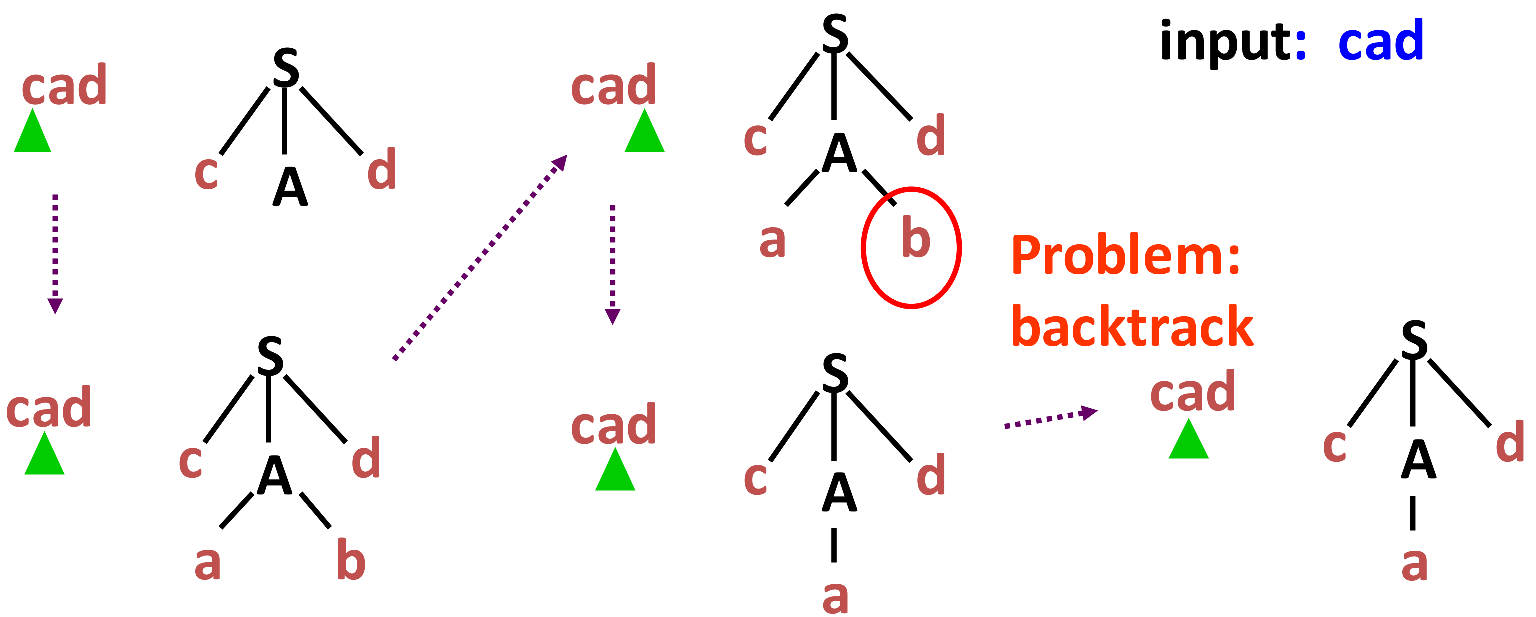
- **Recursive-Descent Parsing**
  - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
  - It is a general parsing technique, but not widely used.
  - Not efficient
- **Predictive Parsing**
  - no backtracking
  - efficient
  - needs a special 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.

$A \Rightarrow aBc \Rightarrow adDc \Rightarrow adec$

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

Recursive-Descent Parsing (uses Backtracking)

- General category of Top-Down Parsing
- Choose production rule based on input symbol
- May require backtracking to correct a wrong choice.
- Example: $S \rightarrow c\ A\ d$
  $A \rightarrow ab \mid a$



Predictive Parser

a grammar ➔ ➔ a grammar suitable for pedictive

eliminate left recursion ; left factor — 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 → **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, $A \rightarrow X_1 X_2 \cdots X_k$;
    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 */;
    }
}

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';

}

}

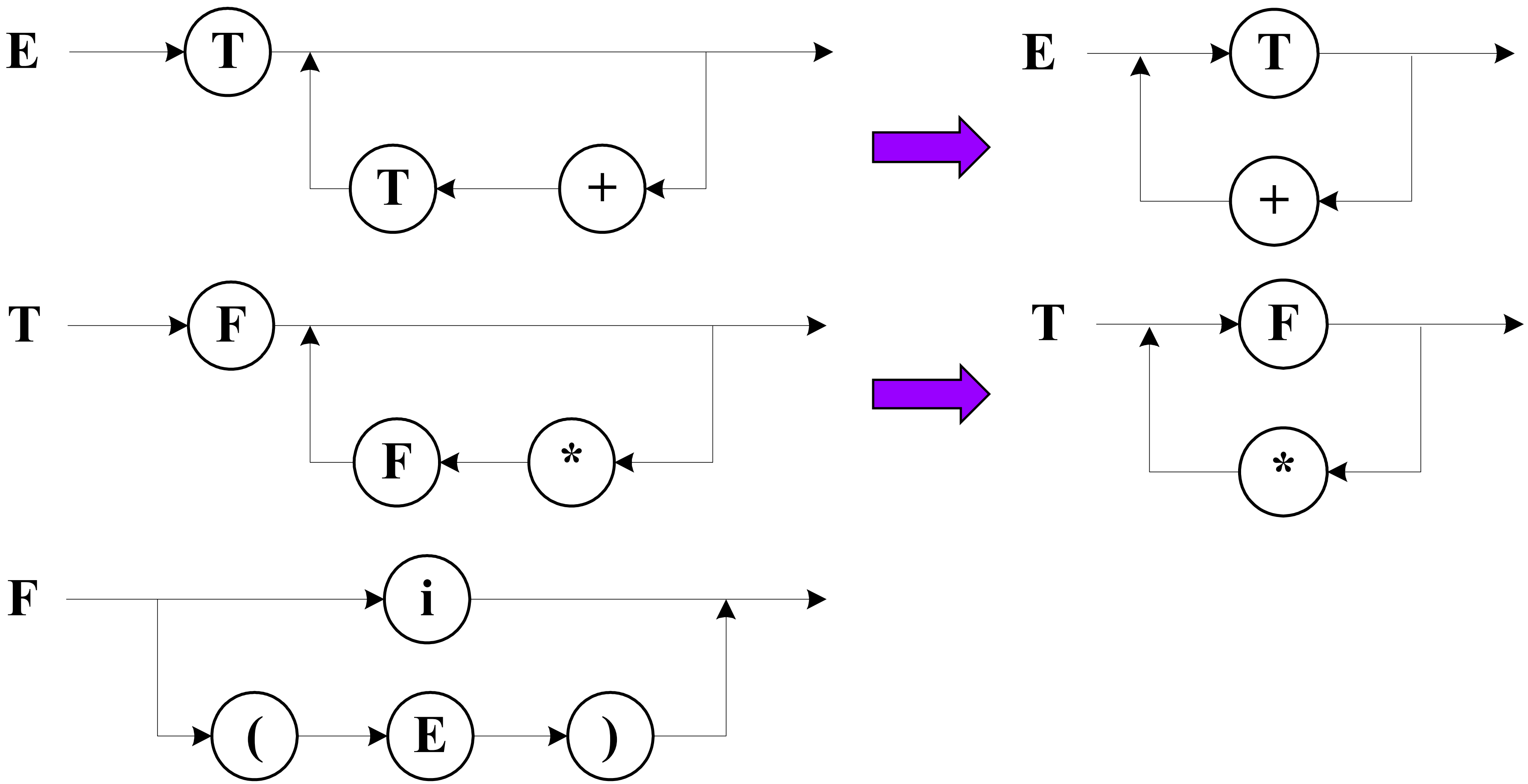
F → (E) | number

```
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: $E \rightarrow T | E+T$
  $T \rightarrow F | T*E$
  $F \rightarrow i | (E)$

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



```
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 → i | (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 → F{∗F}

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

FIRST Set

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

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

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

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

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

FIRST Example

Example 4.17:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | 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}

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:

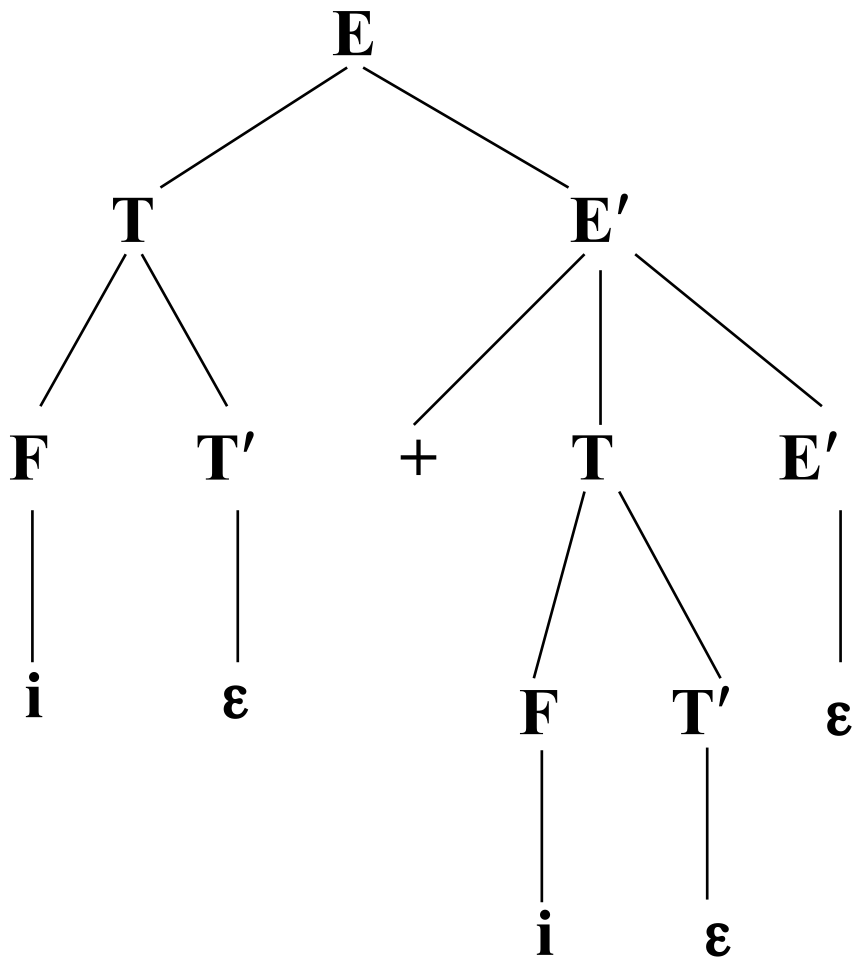
$A \rightarrow aB \mid bC$
$B \rightarrow b \mid dD$
$C \rightarrow c$
$D \rightarrow d$

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

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

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

- $E \rightarrow TE'$

  $T \rightarrow FT'$

  $F \rightarrow i$

  $T' \rightarrow \varepsilon$

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

  $T \rightarrow FT'$

  $F \rightarrow i$

  $T' \rightarrow \varepsilon$

  $E' \rightarrow \varepsilon$

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(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B) .

FOLLOW Example

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

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 $\alpha \rightarrow \in$ or $\alpha \Rightarrow^* \varepsilon$, then what follows A dictates the next choice to be made.**

- **If $A \rightarrow \alpha$, 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.)**

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 $A \to \alpha$ and $A \to \beta$
  - Both α and β cannot derive strings starting with same terminals.

    $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$, $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$ $(1 \leq i \neq j \leq n)$
  - 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 $\varepsilon \in FIRST(\alpha_i) (1 \leq i \leq n)$, then $FIRST(\alpha_i) \cap FOLLOW(A) = \varnothing$

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.

LL(1) Parse Tables

$E \rightarrow i \mid (E\ Op\ E)$ $\qquad$ $Op \rightarrow + \mid *$

V_N	input symbol				
	i	(	)	+	*
E	$E \rightarrow i$	$E \rightarrow E\ Op\ E)$			
E′				$Op \rightarrow +$	$Op \rightarrow *$

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) .

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

- 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

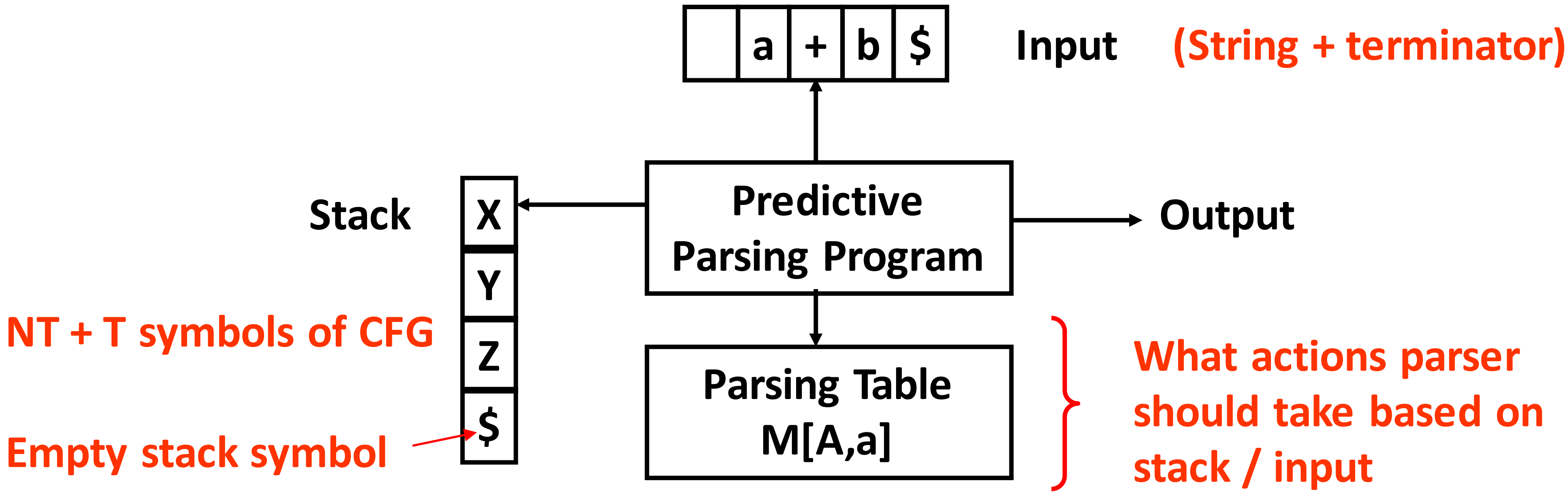
- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

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

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

input : $i_1 * i_2 + i_3$

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

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

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 & 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.
  - $A \rightarrow A\alpha \mid \beta$
    - 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
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - 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.

Examples

- Example: $S \rightarrow c\,A\,d \qquad A \rightarrow ab \mid a$

  Left Factoring: $S \rightarrow c\,A\,d \qquad A \rightarrow aB \qquad B \rightarrow a \mid \varepsilon$

- Example: $S \rightarrow Sa \mid *$

  Eliminate left recursion: $S \rightarrow *B \qquad B \rightarrow aB \mid \varepsilon$

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.

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

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	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S′			$S' \rightarrow eS$ $\cancel{S' \rightarrow \varepsilon}$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				

Error Recovery in Prdictive 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

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

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

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

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

Panic-Mode Error Recovery – Example

$S \rightarrow AbS \mid e \mid \varepsilon$

$A \rightarrow a \mid cAda$

	a	b	c	d	e	$
S	S → AbS	*sync*	S → AbS	*sync*	S → e	S → ε
A	A → a	*sync*	A → cAd	*sync*	*sync*	*sync*

stack	**input**	**output**
$S	ceadb$	S → AbS
$SbA	ceadb$	A → cAd
$SbdAc	ceadb$	
$SbdA	eadb$	Error:unexpected e (illegal A)
	(Remove all input tokens until first b or d, pop A)	
$Sbddb$		
$Sb	b$	
$S	$	S → ε
$	$	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
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu