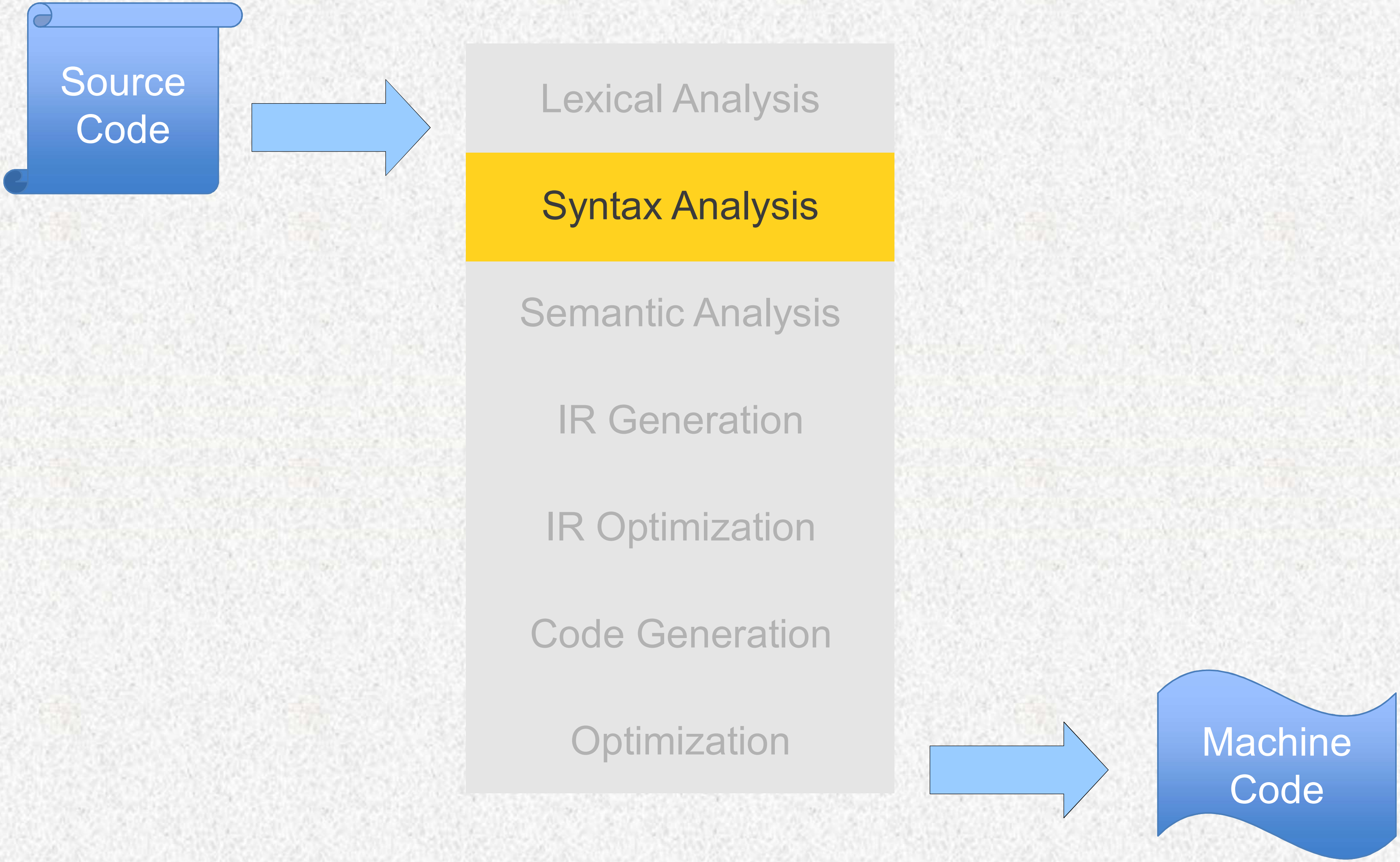
Lecture 6: Syntax Analysis (cont.)

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

Syntax Analysis

Where are we ?


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

Derivations Revisited

- A derivation encodes two pieces of information:
  - What productions were applied to produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

Derivation exercise 1

Productions:

***assign_stmt* $\rightarrow$ *id* := *expr* ;**

expr* $\rightarrow$ *expr op term

expr* $\rightarrow$ *term

term* $\rightarrow$ *id

term* $\rightarrow$ *real

term* $\rightarrow$ *integer

***op* $\rightarrow$ +**

***op* $\rightarrow$ -**

Let's derive:

id := id + real – integer ;

Please use left-most derivation

id := id + real – integer ;

Left-most derivation:

assign_stmt

$\Rightarrow$ *id := expr ;*

$\Rightarrow$ *id := expr op term ;*

$\Rightarrow$ *id := expr op term op term ;*

$\Rightarrow$ *id := term op term op term ;*

$\Rightarrow$ *id := id op term op term;*

$\Rightarrow$ *id := id + term op term ;*

$\Rightarrow$ *id := id + real op term ;*

$\Rightarrow$ *id := id + real - term ;*

$\Rightarrow$ *id := id + real - integer;*

Using production:

assign_stmt $\rightarrow$ *id := expr ;*

expr $\rightarrow$ *expr op term*

expr $\rightarrow$ *expr op term*

expr $\rightarrow$ *term*

term $\rightarrow$ *id*

op $\rightarrow$ *+*

term $\rightarrow$ *real*

op $\rightarrow$ *-*

term $\rightarrow$ *integer*

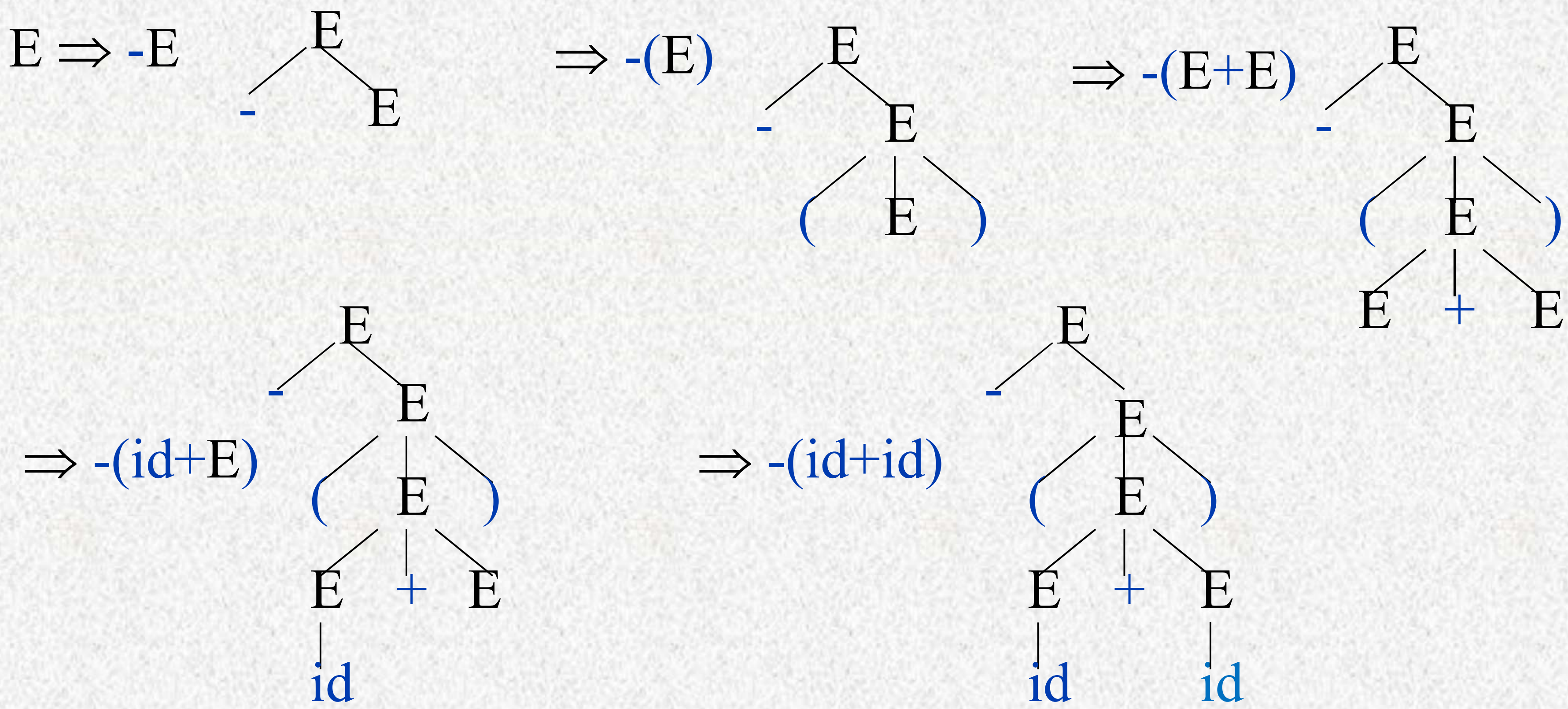
Parse Trees

- A parse tree is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- **A parse tree can be seen as a graphical representation of a derivation.**

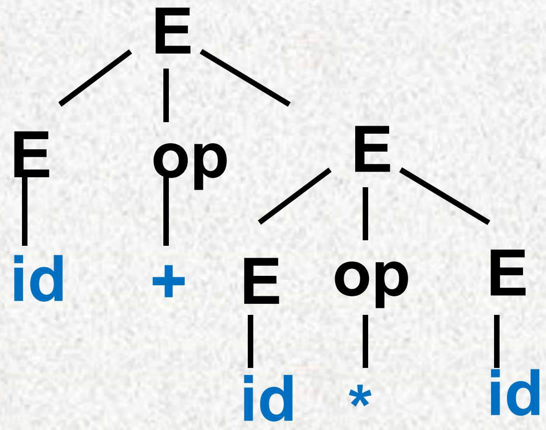
EX. $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$



$E \rightarrow E \text{ op } E \mid (E) \mid -E \mid \text{id}$

$\text{op} \rightarrow + \mid - \mid * \mid /$

$$
\begin{aligned}
E &\Rightarrow E \text{ op } E \\
&\Rightarrow \text{id op } E \\
&\Rightarrow \text{id} + E \\
&\Rightarrow \text{id} + E \text{ op } E \\
&\Rightarrow \text{id} + \text{id op } E \\
&\Rightarrow \text{id} + \text{id} * E \\
&\Rightarrow \text{id} + \text{id} * \text{id}
\end{aligned}
$$



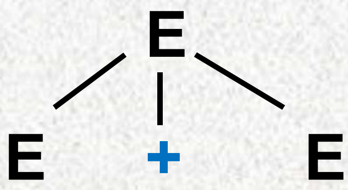
Parse Trees and Derivations

Consider the expression grammar:

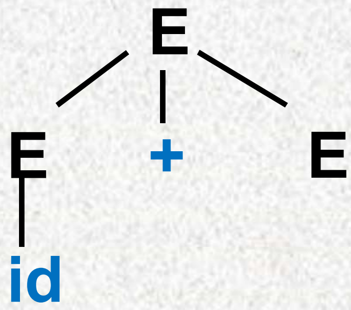
$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$$

Leftmost derivations of $id + id * id$

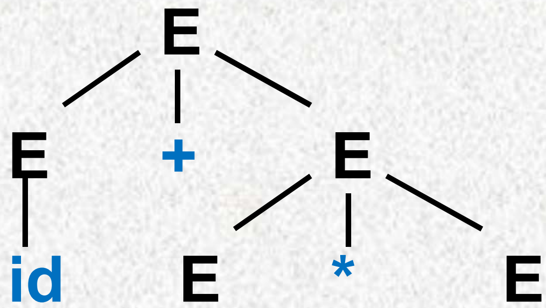
$E \Rightarrow E + E$



$E + E \Rightarrow id + E$

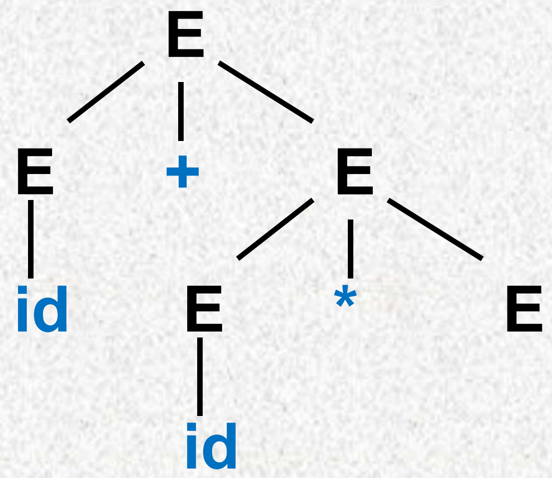


$id + E \Rightarrow id + E * E$

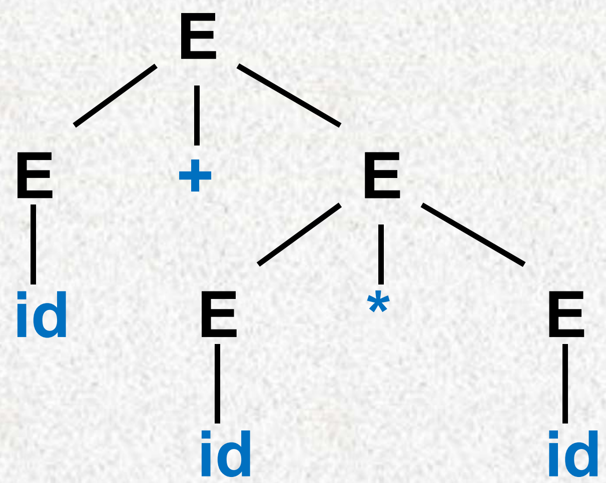


Parse Trees and Derivations (cont.)

$\mathbf{id + E * E \Rightarrow id + id * E}$



$\mathbf{id + id * E \Rightarrow id + id * id}$



Alternative Parse Tree & Derivation

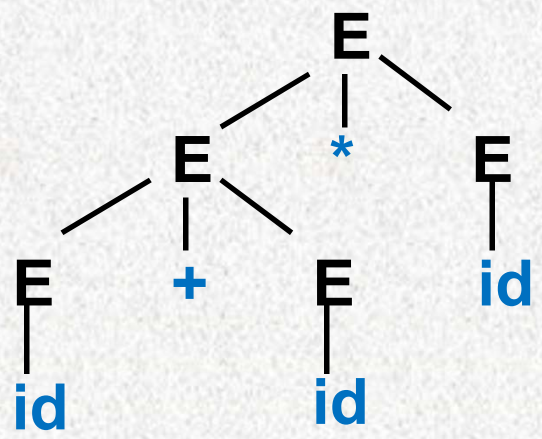
$E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



WHAT'S THE ISSUE HERE ?

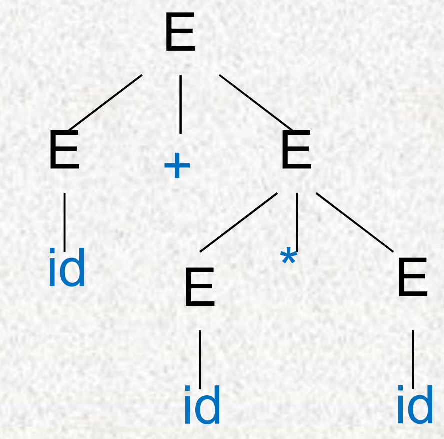
Two distinct leftmost derivations!

Challenges in Parsing

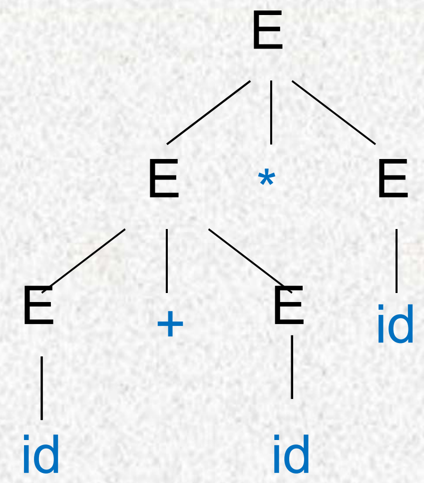
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$



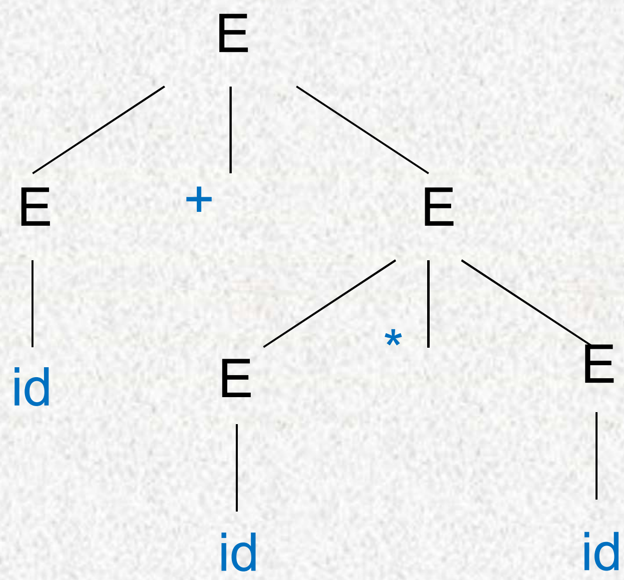
$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$

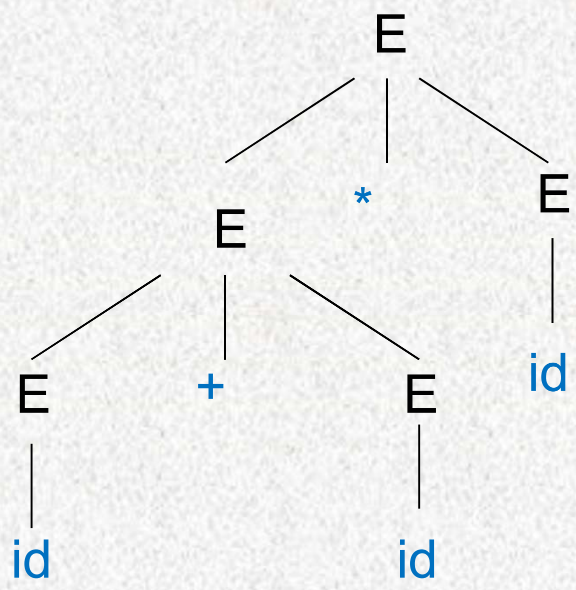


two parse trees for id+id*id.

Is Ambiguity a Problem?

Depends on semantics.





Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through layering.
    - Have exactly one way to build each piece of the string?
    - Have exactly one way of combining those pieces back together?

Resolving Ambiguity

- For the most parsers, the grammar must be unambiguous.
- **unambiguous grammar**
- ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity** rules.

$E \rightarrow E+E \mid E*E \mid E\hat{}E \mid id \mid (E)$

disambiguate the grammar

precedence: ^ (right to left)

* (left to right)

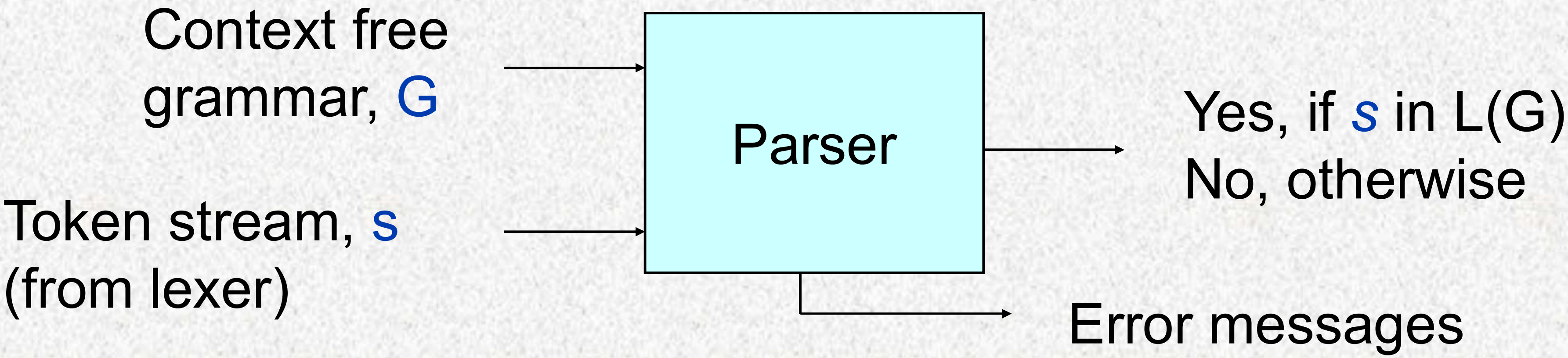
\+ (left to right)

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow G\hat{}F \mid G$
$G \rightarrow id \mid (E)$

Rewrite to eliminate the ambiguity

Or, simply tell which parse tree should be selected

A Parser



- Syntax analyzers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted

- Various kinds: **LL(k), LR(k), SLR, LALR**

Types

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

Homework

Page 206: Exercise 4.2.1
Page 207: Exercise 4.2.2 (d) (f) (g)

Top-Down Parsing

Two Key Points

expression	$\rightarrow$	*expression* + *term*
expression	$\rightarrow$	*expression* - *term*
expression	$\rightarrow$	*term*
term	$\rightarrow$	*term* * *factor*
term	$\rightarrow$	*term* / *factor*
term	$\rightarrow$	*factor*
factor	$\rightarrow$	(*expression*)
factor	$\rightarrow$	**id**

expression => term
=> term*factor
=> term/factor*factor

– Q1: Which non-terminal to be replaced?

Leftmost derivation $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$

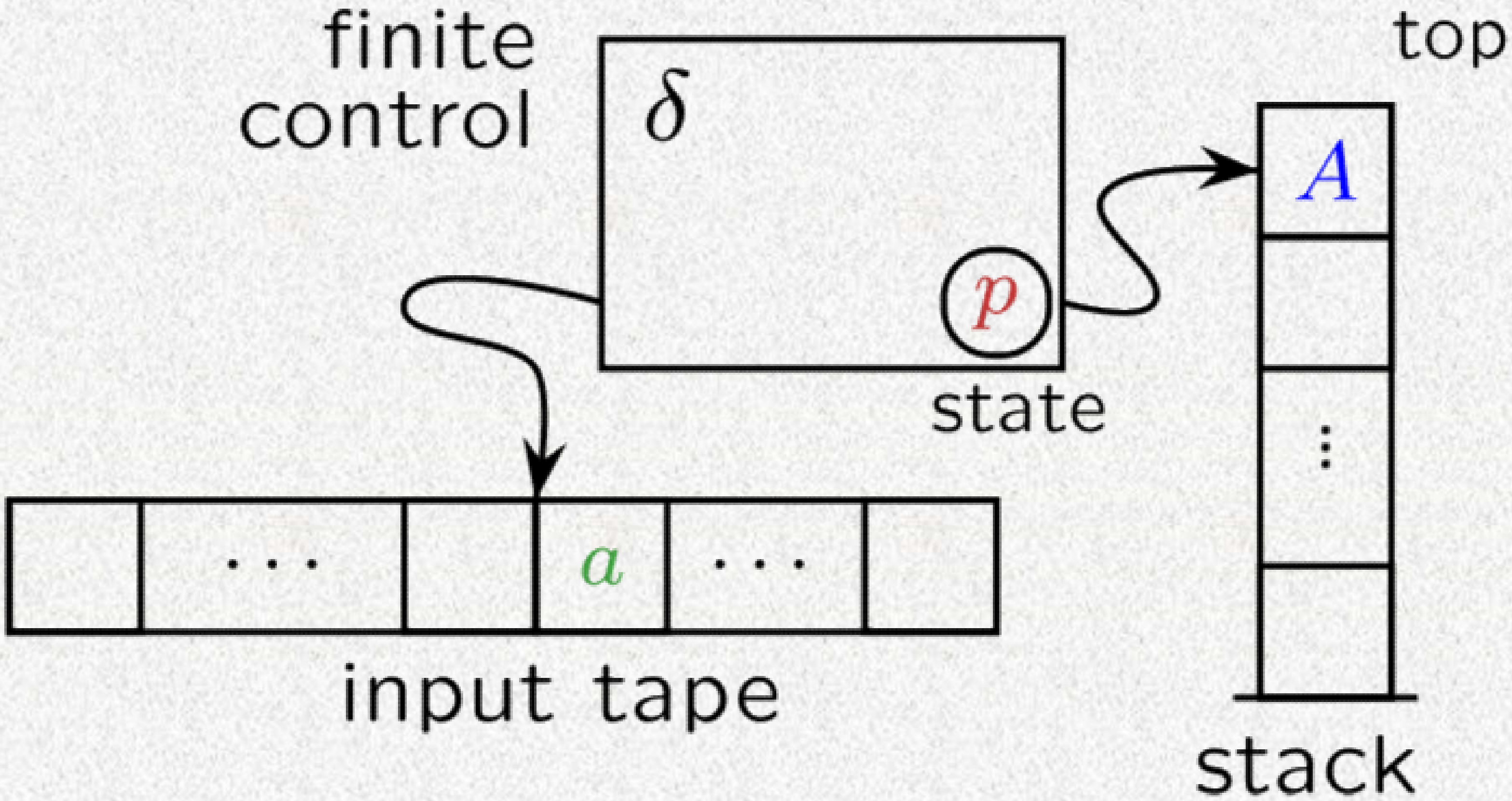
– Q2: Which production to be used?

Top-Down Parsing

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

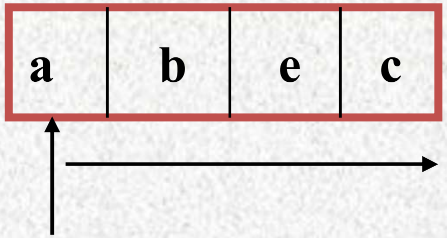
By always replacing the leftmost non-terminal symbol via a production rule, we are guaranteed of developing a parse tree in a left-to-right fashion that is consistent with scanning the input.

Pushdown Automaton



An illustration with PDA

P:
(1) $Z \rightarrow aBeA$
(2) $A \rightarrow Bc$
(3) $B \rightarrow d$
(4) $B \rightarrow bB$
(5) $B \rightarrow \varepsilon$



Reading Head	Stack	Analysis	Derivation	Match?
abec	Z	Z production starting with a? - (1)	aBeA	a
bec	BeA	B production starting with b? –(4)	bBeA	b
ec	BeA	B production starting with e? -(5)	εeA	e
c	A	A production starting with c? -(2)(5)		

An illustration with PDA

P:
(1) $Z \rightarrow aBeA$
(2) $A \rightarrow Bc$
(3) $B \rightarrow d$
(4) $B \rightarrow bB$
(5) $B \rightarrow \varepsilon$

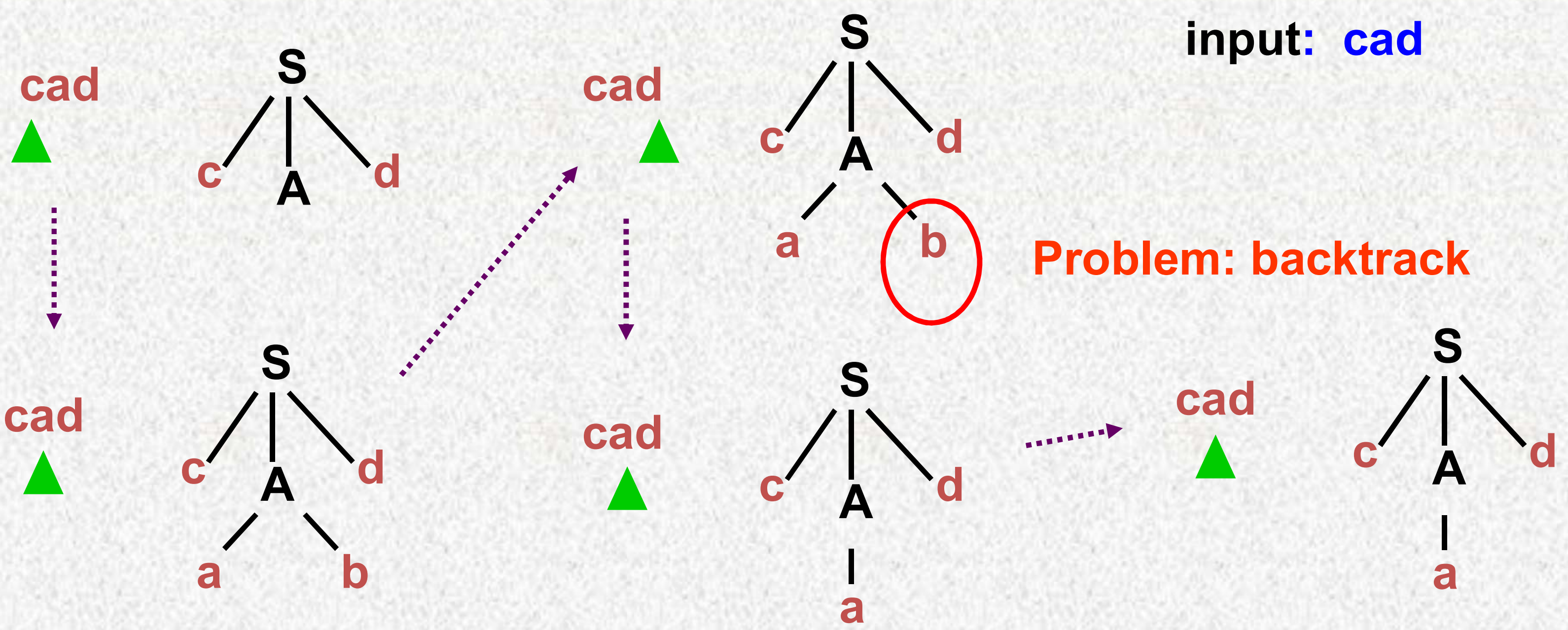
Reading Head	Stack	Analysis	Derivation	Match?
c	A	A production starting with c?-(2)	Bc	
c	Bc	A production starting with c? －(5)	εc	c

a	**b**	**e**	**c**

Problem - Backtraking

- General category of Top-Down Parsing
- Choose production rule based on input symbol
- May require backtracking to correct a wrong choice.

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

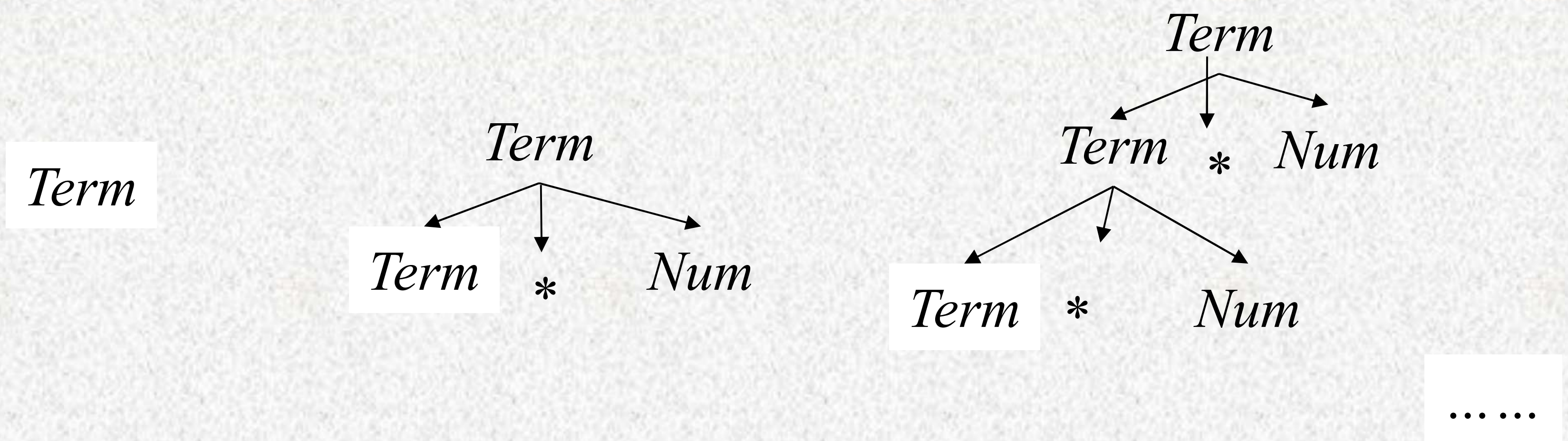


Problem – Left recursion

- A grammar is Left Recursion if it has a nonterminal A such that there is a derivation A $\Rightarrow^+$ Aα for some string α.

Left Recursion + top-down parsing = infinite loop

Eg. Term $\rightarrow$ Term*Num



Elimination of Left recursion

- Eliminating Direct Left Recursion

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$\beta_i \alpha_i^*$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

Elimination of Left recursion

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

  elimination of left recursion

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

- $P \rightarrow P\alpha_1 \mid P\alpha_2 \mid \ldots \mid P\alpha_m^1 \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- elimination of left recursion

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

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

Elimination of Left recursion (eg.)

- G[E]: $E \rightarrow E+T|T$

  $T \rightarrow T*F|F$

  $F \rightarrow (E)|\ I$

Elimination of Left Recursion

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

Elimination of Left recursion (eg.)

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

P→ PaPb|BaP

- We have α = aPb, β=BaP
- So, P→ βP'

  P'→ αP'|ε
- 改写后：P→ BaPP'

  P'→ aPbP'|ε

Multiple P? Consider the most-left one.

Elimination of Indirect Left recursion

Direct： $S \rightarrow Sa$

Indirect： $S \rightarrow Aa, A \overset{+}{\rightarrow} Sb$, then we have $A \overset{+}{\rightarrow} Aab$

e.g： S → Aa | b,   A → Sd |ε
1

S => Aa => Sda

Elimination of Left recursion algorithm

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A_1, A_2, ..., A_n.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form A_i → A_j γ by the
              productions A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ, where
              A_j → δ_1 | δ_2 | ··· | δ_k are all current A_j-productions
5)      }
6)      eliminate the immediate left recursion among the A_i-productions
7)  }
```

Elimination of Left recursion (eg.)

$S \rightarrow A\ b$
$A \rightarrow S\ a \mid b$

1:S
2:A

$A \rightarrow Aba \mid b$

$A \rightarrow bA'$
$A' \rightarrow baA' \mid \varepsilon$

Elimination of Left recursion (eg.)

$S \rightarrow Aa \mid b,$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

1:S
2:A

$S \rightarrow Aa \mid b,$
$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

$S \rightarrow Aa \mid b,$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Elimination of Left recursion (eg.)

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

1:S
2:Q
3:R

$S \rightarrow Qc \mid c$
$Q \rightarrow Rb \mid b$
$R \rightarrow Sa \mid a$
$\rightarrow (Qc|c)a \mid a$
$\rightarrow Qca \mid ca \mid a$
$\rightarrow (Rb|b)ca \mid ca \mid a$

$S \rightarrow Qc \mid c$
$Q \rightarrow Rb \mid b$
$R \rightarrow (bca \mid ca \mid a)R'$
$R' \rightarrow bcaR' \mid \varepsilon$

Elimination of Left recursion (eg.)

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

1:R
2:Q
3:S

$R \rightarrow Sa \mid a$
$Q \rightarrow Rb \mid b \rightarrow Sab \mid ab \mid b$
$S \rightarrow Qc \mid c \rightarrow Sabc \mid abc \mid bc \mid c$

$S \rightarrow (abc \mid bc \mid c)S'$
$S' \rightarrow abcS' \mid \varepsilon$

Problem - Left Factoring

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

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

- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

  $A \rightarrow \alpha A' \mid \gamma$

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

Problem - Left Factoring

Algorithm 4.21: Left factoring a grammar.

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. □

Problem - Left Factoring

- E.g
  - $S \rightarrow iEtS \mid iEtSeS \mid a$

    $E \rightarrow b$
  - For, S, the longest pre-fix is *iEtS*, Thus,

    $S \rightarrow iEtSS' \mid a$

    $S' \rightarrow eS \mid \varepsilon$

    $E \rightarrow b$

Problem - Left Factoring

- E.g.

G：

(1) $S \rightarrow aSb$

(2) $S \rightarrow aS$

(3) $S \rightarrow \varepsilon$

For (1)、(2)， extract the left factor：

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

$S \rightarrow \varepsilon$

We have G′：

$S \rightarrow aSA$

$A \rightarrow b$

$A \rightarrow \varepsilon$

$S \rightarrow \varepsilon$

Homework

Page 216: Exercise 4.3.1

Two Parsing Methods

A Naïve Method

- **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

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-Descent Parsing

- Example

P:
(1) $Z \rightarrow aBd$ {a}
(2) $B \rightarrow d$ {d}
(3) $B \rightarrow c$ {c}
(4) $B \rightarrow bB$ {b}

a b c d

```
Z ( )
{
if (token == a)
 {   match(a);
      B( );
      match(d);
  }
else error();
}
```

```
B ( )
{
 case token of
  d:  match(d);break;
  c:   match(c); break;
  b:{  match(b);
         B( ); break;}
  other: error();
}
```

```
void main()
{read();
  Z();  }
```

A Non-Recursive Method

- **Predictive Parsing**
  - no backtracking, efficient
  - needs a special form of grammars (**LL(1) grammars**).
  - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

A Non-Recursive Method

- Predict($A \rightarrow \alpha$)
- First(α)
- Follow(A)

FIRST Set

FIRST(α)

If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set:

1. If X is terminal, then FIRST(X) is {X}.

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is nonterminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$. is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if $Y_1 \Rightarrow \varepsilon$, then we add FIRST($Y_2$) and so on.

Now, we can compute FIRST for any string $X_1X_2 \ldots X_n$ as follows. Add to FIRST($X_1X_2 \ldots X_n$) all the non-ε symbols of FIRST(X_1). Also add the non-ε symbols of FIRST(X_2) if ε is in FIRST(X_1), the non-ε symbols of FIRST(X_3) if ε is in both FIRST(X_1) and FIRST(X_2), and so on. Finally, add ε to FIRST($X_1X_2 \ldots X_n$) if, for all i, FIRST(X_i) contains ε.

FIRST Example

- First(α)

First(E'T'E) =?
First(T'E') = ?

E	{i, n , (}
E'	{ + , ε }
T	{ i, n , (}
T'	{ *, ε }
F	{ i, n , (}

P:
(1) $E \rightarrow TE'$
(2) $E' \rightarrow + TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow * F T'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow i$
(9) $F \rightarrow n$

Sε = {E', T'}

First(E'T'E) = {+,*,i,n,(}
First(T'E') = {+,*,ε}

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:

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

FOLLOW Set

Define FOLLOW(A), for nonterminal A, to be the set of terminals *a* that can appear immediately to the right of A in some sentential form, that is, the set of terminals *a* such that there exists a derivation of the form $S \Rightarrow \alpha A a \beta$ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and *a*, but if so, they derived ε and disappeared. If A can be the rightmost symbol in some sentential form, then \$, representing the input right endmarker, is in 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:

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.

2. If there is a production $A \Rightarrow \alpha B\beta$, then everything in FIRST(β), except for ε, is placed in FOLLOW(B).

3. If there is a production $A \Rightarrow \alpha B$, or a production $A \Rightarrow \alpha B\beta$ where FIRST(β) contains ε (i.e., $\beta \Rightarrow \varepsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

FOLLOW Set Example

P:

(1) $E \rightarrow TE'$
(2) $E' \rightarrow +TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow *FT'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow i$
(9) $F \rightarrow n$

First(X)

E	{i, n , (}
E'	{ + , ε }
T	{ i, n , (}
T'	{ *, ε }
F	{ i, n , (}

Follow(X)

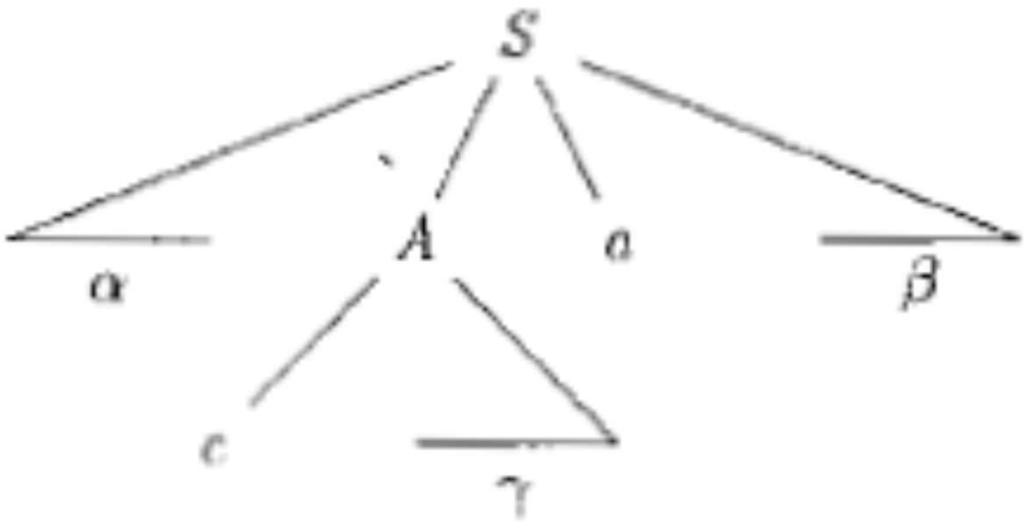
E	{#,)}
E'	{#,)}
T	{+,), #}
T'	{+,), #}
F	{*, +,), #}

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.)

Motivation Behind FOLLOW

$S \Rightarrow^* \alpha A a \beta$



a is in Follow(A); c is in First(A)

Predict Set

- Predict($A \rightarrow \alpha$)
  - Predict($A \rightarrow \alpha$) = First(α), if $\varepsilon \notin$ First(α);
  - Predict($A \rightarrow \alpha$) = First(α)- $\{\varepsilon\} \cup$ Follow(A), if $\varepsilon \in$ First(α);

Predict Set Example

P:

(1) $E \rightarrow TE'$ → First(TE')={i, n,(}

(2) $E' \rightarrow + TE'$ → First(+TE')={+}

(3) $E' \rightarrow \varepsilon$ → Follow(E')={#,)}

(4) $T \rightarrow FT'$ → First(FT')={i,n,(}

(5) $T' \rightarrow * F T'$ → First(*FT')={*}

(6) $T' \rightarrow \varepsilon$ → Follow(T')={),+, # }

(7) $F \rightarrow (E)$ → First((E))={ (}

(8) $F \rightarrow i$ → First(i)={i}

(9) $F \rightarrow n$ → First(n)={n}

first

E	{i, n , (}
E'	{ + , ε }
T	{ i, n , (}
T'	{ *, ε }
F	{ i, n , (}

Follow

E	{#,)}
E'	{#,)}
T	{+,), #}
T'	{+,), #}
F	{*, +,), #}

Now We consider LL(1)

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.

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

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.

Predictive Parser

a grammar ➔ ➔ a grammar suitable for predictive
eliminate left parsing (a LL(1) grammar)
left recursion factor 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

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: $S \rightarrow c\ A\ d \qquad A \rightarrow aa \mid a$

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

- Example： $S \rightarrow Sa \mid *$

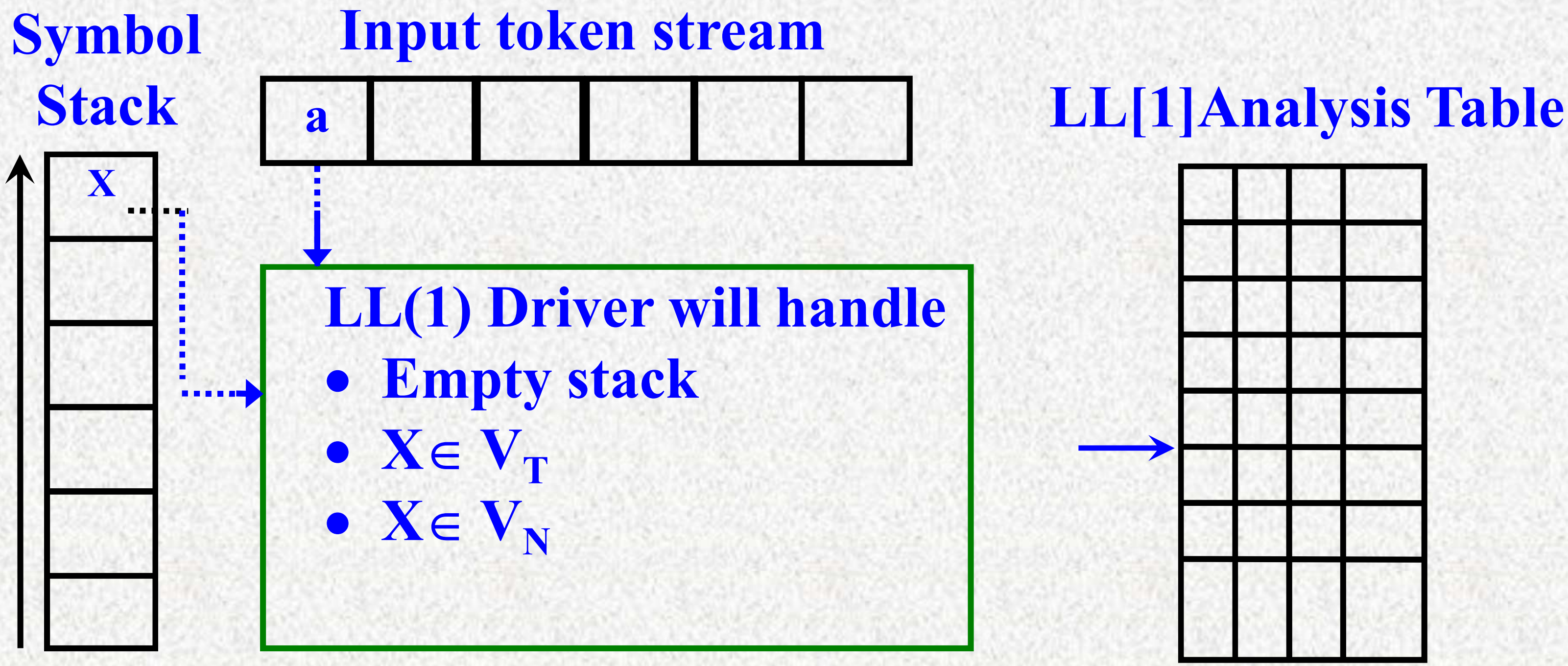
  Eliminate left recursion: $S \rightarrow *B \qquad B \rightarrow aB \mid \varepsilon$

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.

LL(1): a Predictive Parser

LL(1)



- Symbol stack is used to store the intermeddle results for analysis
- When reaching the end of input stream; meanwhile the stack is empty, the string is accepted.
- LL(1) Analysis Table: T(A,a) indicates which production should be used for derivation.

LL(1) Analysis Table

	a_1	…	a_n	#
A_1				
…	….	….	…	
Am				

For LL(1) grammar $G = (V_N, V_T, S, P)$

$V_T = \{a1, \ldots, an\}$, $V_N = \{A1, \ldots, Am\}$

$LL(Ai, aj) = Ai \rightarrow \alpha$, if $aj \in predict(Ai \rightarrow \alpha)$

$LL(Ai, aj) = error(\perp)$, if aj does not belong to any predict($Ai \rightarrow \alpha$)

LL(1) Analysis Table

- Example 1

P:
(1) Z → aBd
(2) B → d
(3) B → c
(4) B → bB

Production	Predict
(1)	{a}
(2)	{d}
(3)	{c}
(4)	{b}

	a	b	c	d	#
Z	(1)				
B		(4)	(3)	(2)	

LL(1) Analysis Table

- Example 2:

(1)	E → TE’	{ i, n, (}
(2)	E’ → + TE’	{+}
(3)	E’ → ε	{#,)}
(4)	T → FT’	{i,n,(}
(5)	T’ → * F T’	{*}
(6)	T’ → ε	{),+, # }
(7)	F → (E)	{ (}
(8)	F → i	{i}
(9)	F → n	{n}

	+	*	(	)	i	n	#
E			(1)		(1)	(1)	
E’	(2)			(3)			(3)
T			(4)		(4)	(4)	
T’	(6)	(5)		(6)			(6)
F			(7)		(8)	(9)	

LL(1) Driver

METHOD: Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above $.

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

A complete example

- LL1-example.pdf

Homework

Page 231: Exercise 4.4.1 (b) (d)
Exercise 4.4.3

Homework

Given a grammar G(T), whose productions are:

T → a[L] | a
L → LL | T

Where ‘a’ ‘[‘ ‘]’ are terminal, T and L are non-terminal. T is the starting symbol.

(1) Please write down a left-most derivation for sentence “a[aa]”

(2) Try to eliminate the left-recursion and left factor (let’s denote the new grammar after this elimination as G’).

(3) For G’, computer the First and Follow set of all non-terminal symbols;

(4) Construct LL(1) parsing table, tell whether the new grammar G’ is LL(1) or not.

(5) Write down the process for analyzing “a[a]” with your LL(1) table.