# Plan for Today

**Logistics**

- Midterm, TUESDAY in class. Examples online. HW3. 1-side 8.5x11" note sheet. Will be placing people in seats randomly.
- PA1 peer review due tonight
- HW3, due SUNDAY night. NO LATE period.
- PA2 partners policy

**Haskell Guards**

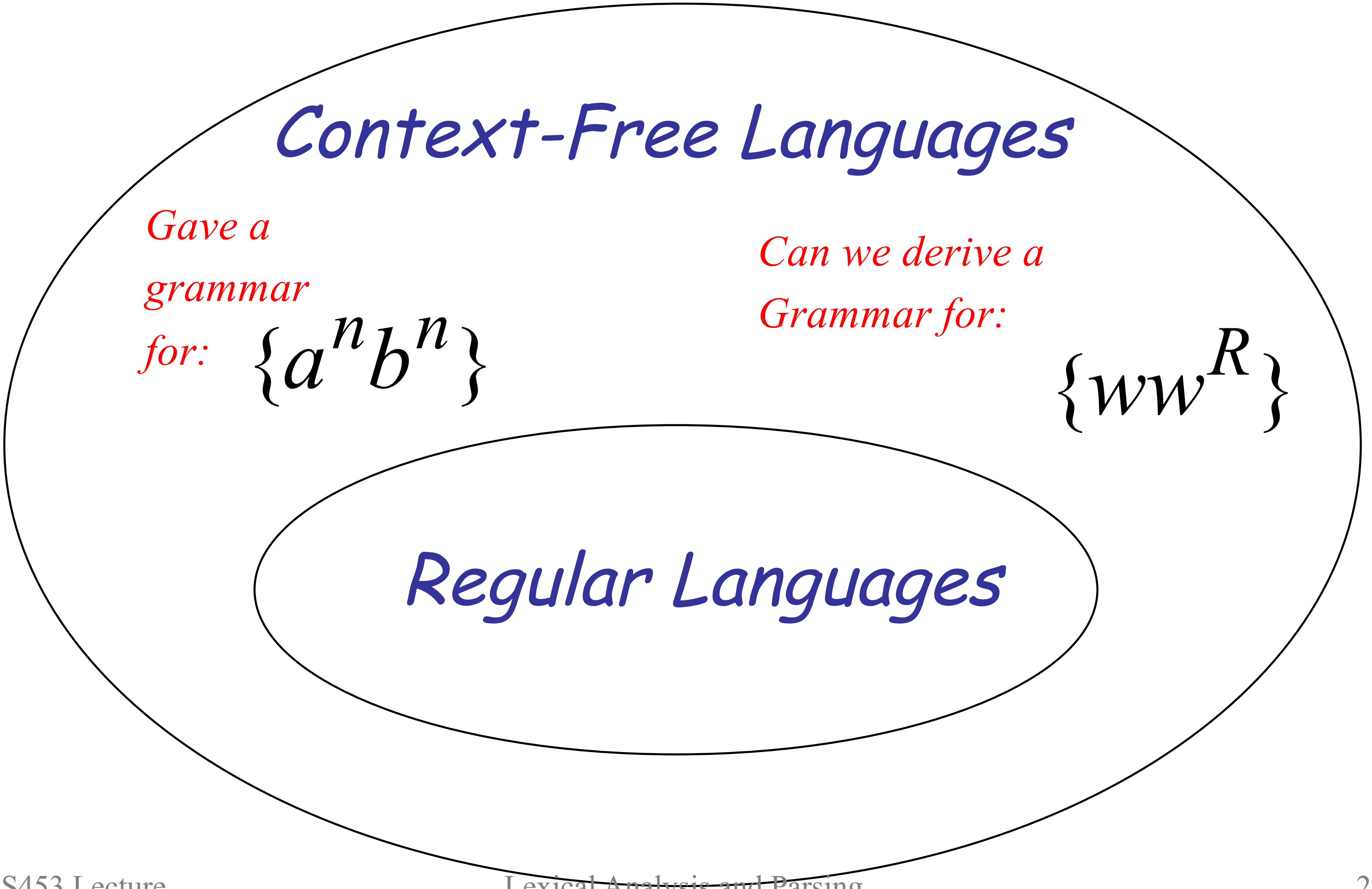
- Useful in the context of the lexer and parser.
- See Mr. Mitchell's slides on Resources page, slide 96 through 99

**Context Free Grammars**

- Derivations
- Parse trees

**Top-down Predictive Parsing**

# Deriving another grammar



Context-Free Languages

Gave a grammar for: $\{a^n b^n\}$

Can we derive a Grammar for: $\{ww^R\}$

Regular Languages

# Example

A context-free grammar $G$ :

$$S \to aSa$$

$$S \to bSb$$

$$S \to \varepsilon$$

A derivation:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

Another derivation:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

# Representing All Properly Nested Parentheses

$$S \to aSb$$

$$S \to \varepsilon$$

$$L(G) = \{a^n b^n : n \geq 0\}$$

Describes parentheses: (((( ))))

*Can we build a grammar to include any valid combination of ( )? For example ( ( ) ( ( ) ) )*

# A Possible Grammar

A context-free grammar $G$ :

$$S \rightarrow (S)$$

$$S \rightarrow SS$$

$$S \rightarrow \varepsilon$$

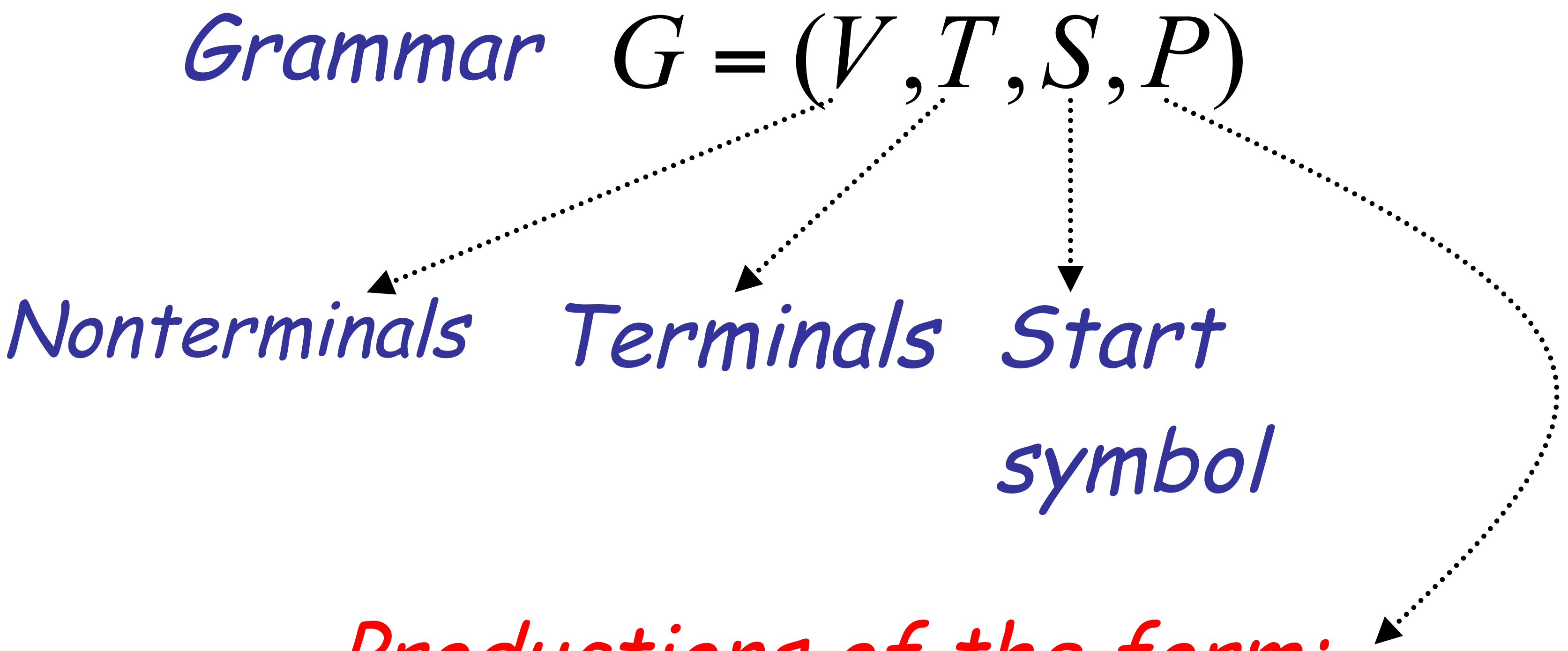
A derivation:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()$$

Another derivation:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

## Context-Free Grammars



Grammar $G = (V, T, S, P)$

Nonterminals

Terminals

Start symbol

Productions of the form:

$$A \rightarrow x$$

Nonterminal

String of symbols, Nonterminals and terminals

# Derivation, Language

**Grammar: G=(V,T,S,P)**

**Derivation:**

**Start with start symbol S**

**Keep replacing non-terminals A by their RHS x, until no non-terminals are left**

**The resulting string (sentence) is part of the language L(G)**

**The Language L(G) defined by the CFG G:**

**L(G) = the set of all strings of terminals that can be derived this way**

## Derivation Order

*Given a grammar with rules:*

1. $S \to AB$

2. $A \to aaA$

3. $A \to \varepsilon$

4. $B \to Bb$

5. $B \to \varepsilon$

*Always expand the leftmost non-terminal*

**Leftmost derivation:**

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

# Derivation Order

*Given a grammar with rules:*

$1.\ S \rightarrow AB$

$2.\ A \rightarrow aaA$

$3.\ A \rightarrow \varepsilon$

$4.\ B \rightarrow Bb$

$5.\ B \rightarrow \varepsilon$

*Always expand the rightmost non-terminal*

## Rightmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

| Grammar | String |
|---|---|
| Stm --> id := Exp | a := ( b := ( c := 3, 2 ), 1 ) |
| Exp --> num | |
| Exp --> ( Stm, Exp ) | |

## *Leftmost derivation:*

Stm ==> a := Exp ==> a := ( Stm, Exp ) ==>  a := ( b := Exp, Exp )
==>  a := ( b := ( Stm, Exp ), Exp ) ==>  a := ( b := ( c := Exp, Exp ), Exp )
==>  a := ( b := ( c := 3, Exp ), Exp ) ==>  a := ( b := ( c := 3, 2), Exp )
==>  a := ( b := ( c := 3, 2), 1)

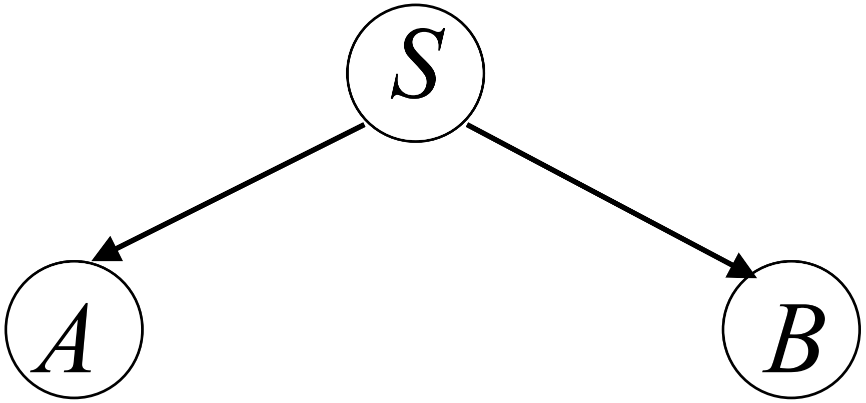
## *Rightmost derivation:*

Stm ==> a := Exp ==> a := ( Stm, Exp ) ==>  a := ( Stm, 1)
==>

# Parse Trees

$S \rightarrow AB \qquad A \rightarrow aaA \mid \varepsilon \qquad B \rightarrow Bb \mid \varepsilon$

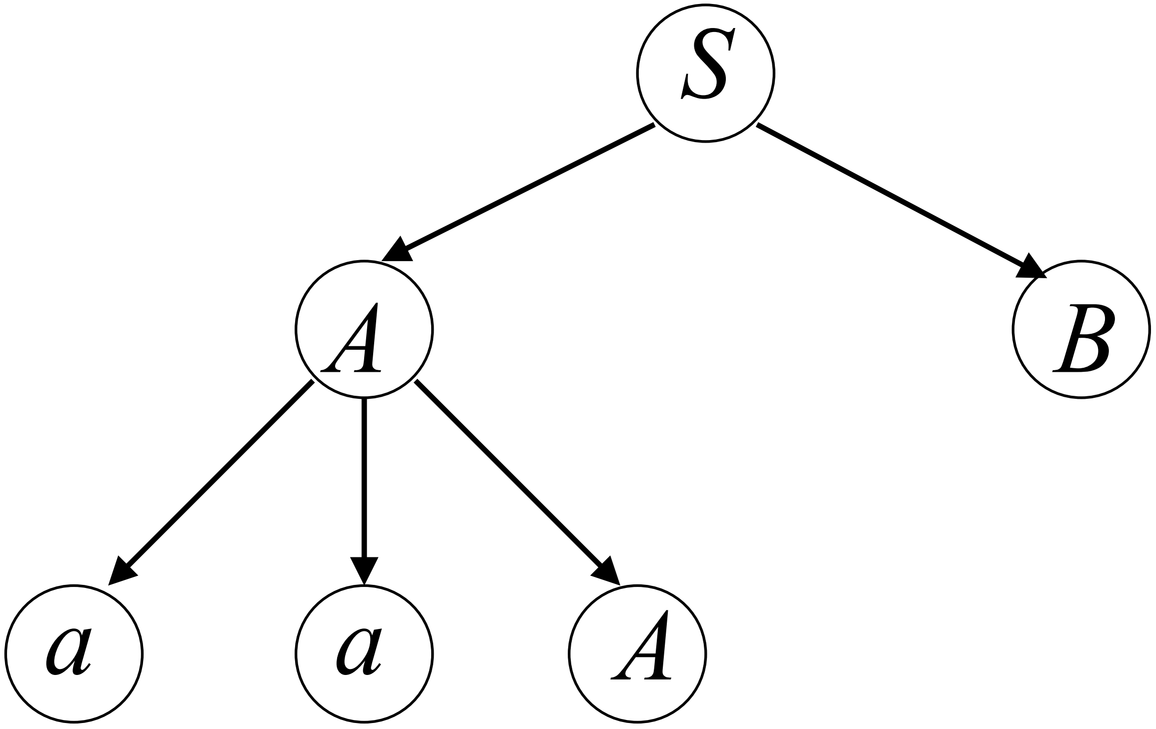
$$S \Rightarrow AB$$



# Parse Trees

$S \rightarrow AB \qquad A \rightarrow aaA \mid \varepsilon \qquad B \rightarrow Bb \mid \varepsilon$

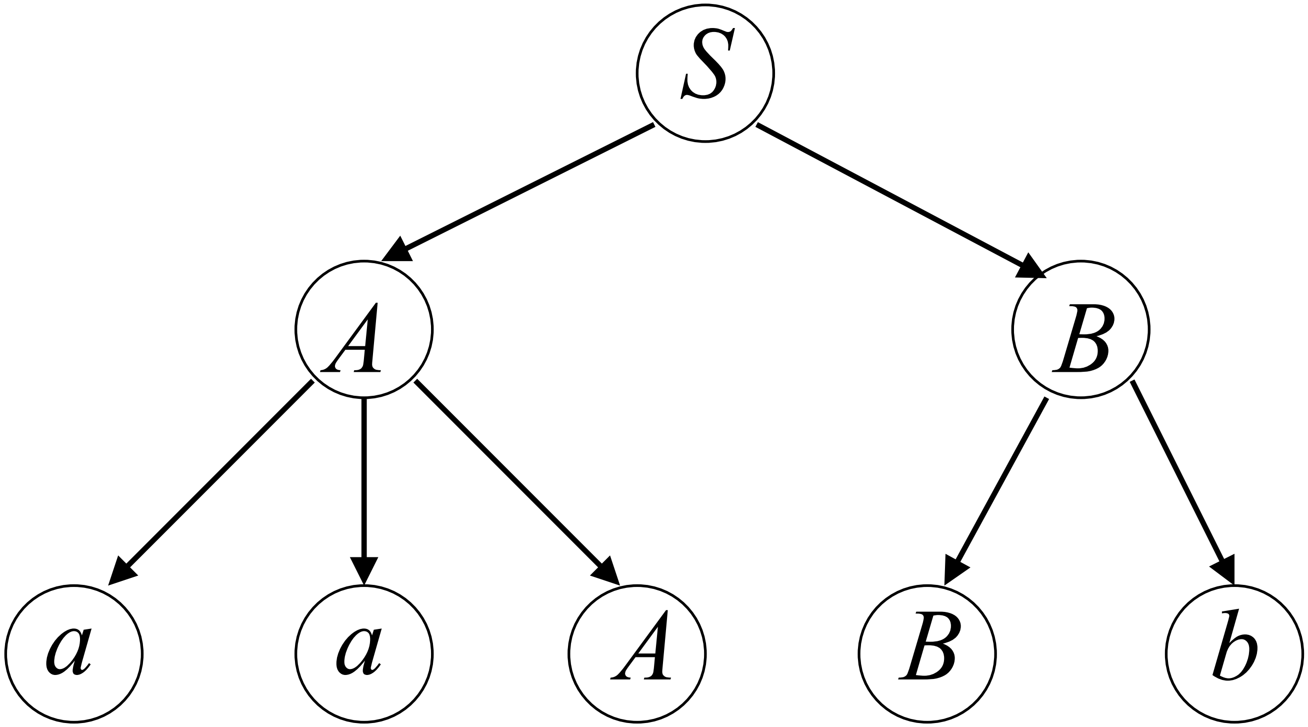
$$S \Rightarrow AB \Rightarrow aaAB$$



# Parse Trees

$S \rightarrow AB \qquad A \rightarrow aaA \mid \varepsilon \qquad B \rightarrow Bb \mid \varepsilon$

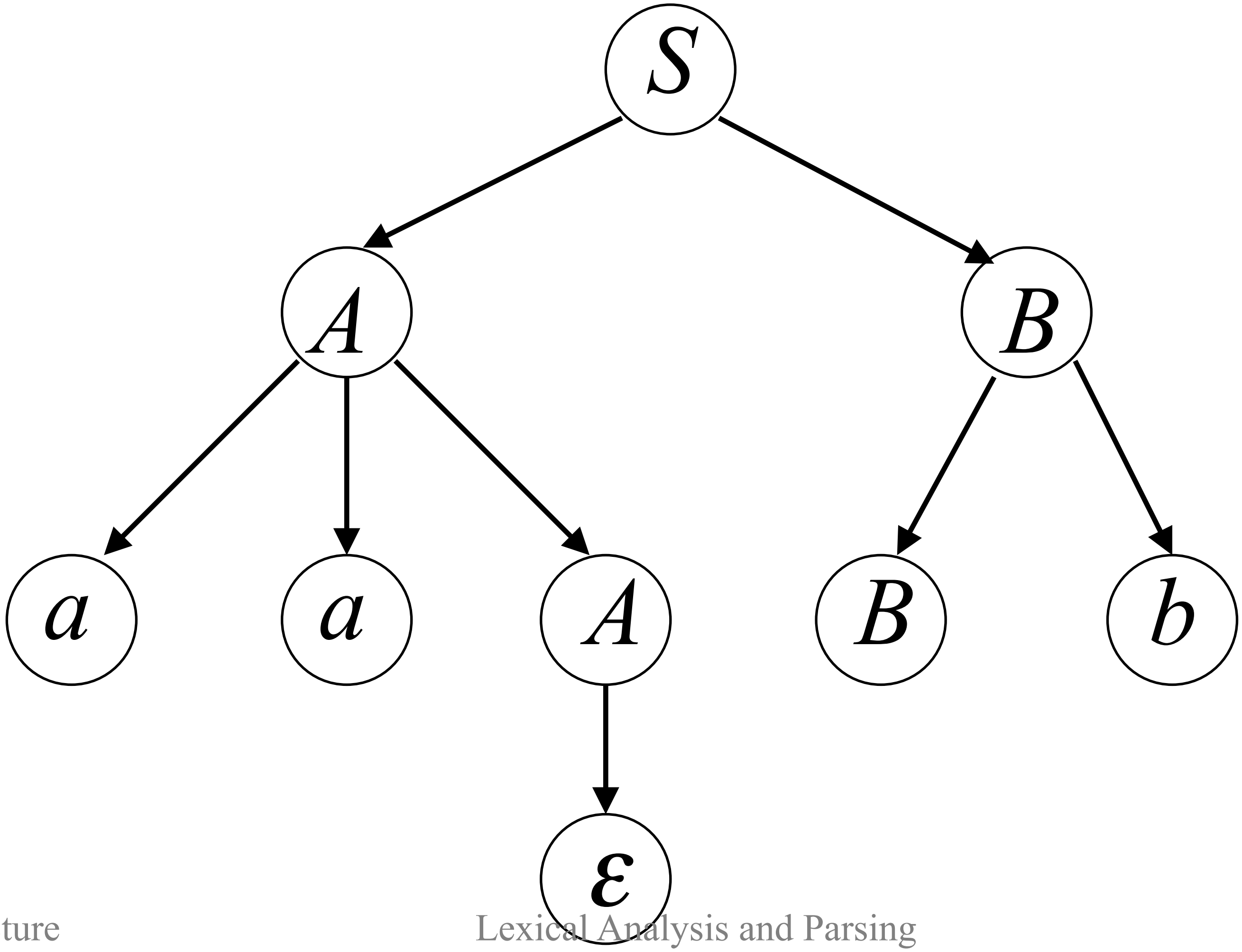
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



# Parse Trees

$S \rightarrow AB \qquad A \rightarrow aaA \mid \varepsilon \qquad B \rightarrow Bb \mid \varepsilon$

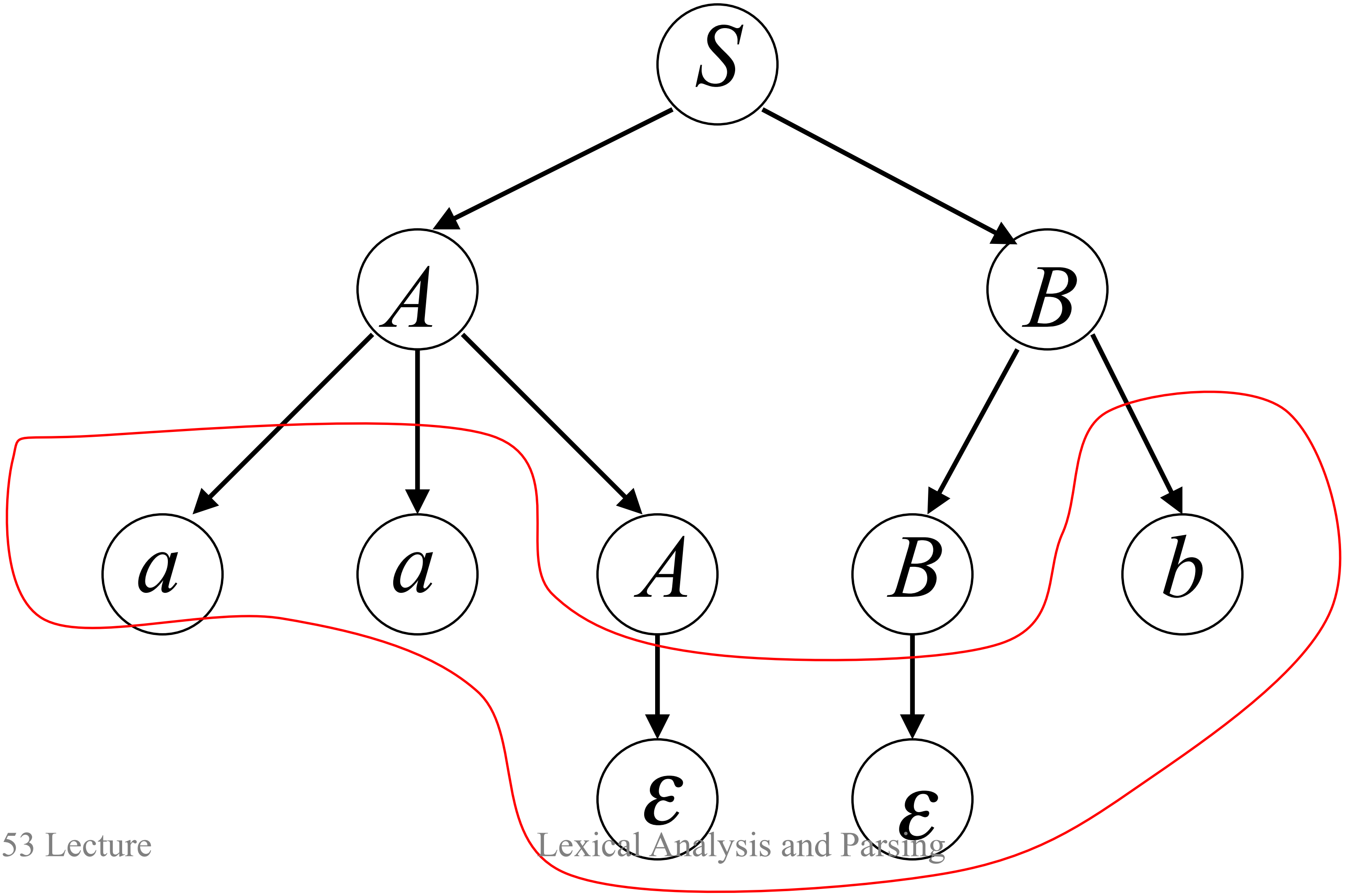
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



# Parse Trees

$S \rightarrow AB \qquad A \rightarrow aaA \mid \varepsilon \qquad B \rightarrow Bb \mid \varepsilon$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$



yield

$aa\varepsilon\varepsilon b$

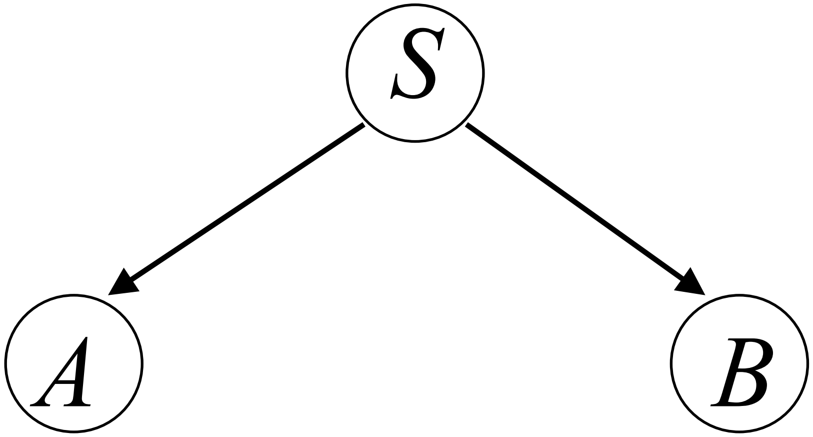
$= aab$

## Sentential forms

$S \rightarrow AB \qquad A \rightarrow aaA \mid \varepsilon \qquad B \rightarrow Bb \mid \varepsilon$

$$S \Rightarrow AB$$

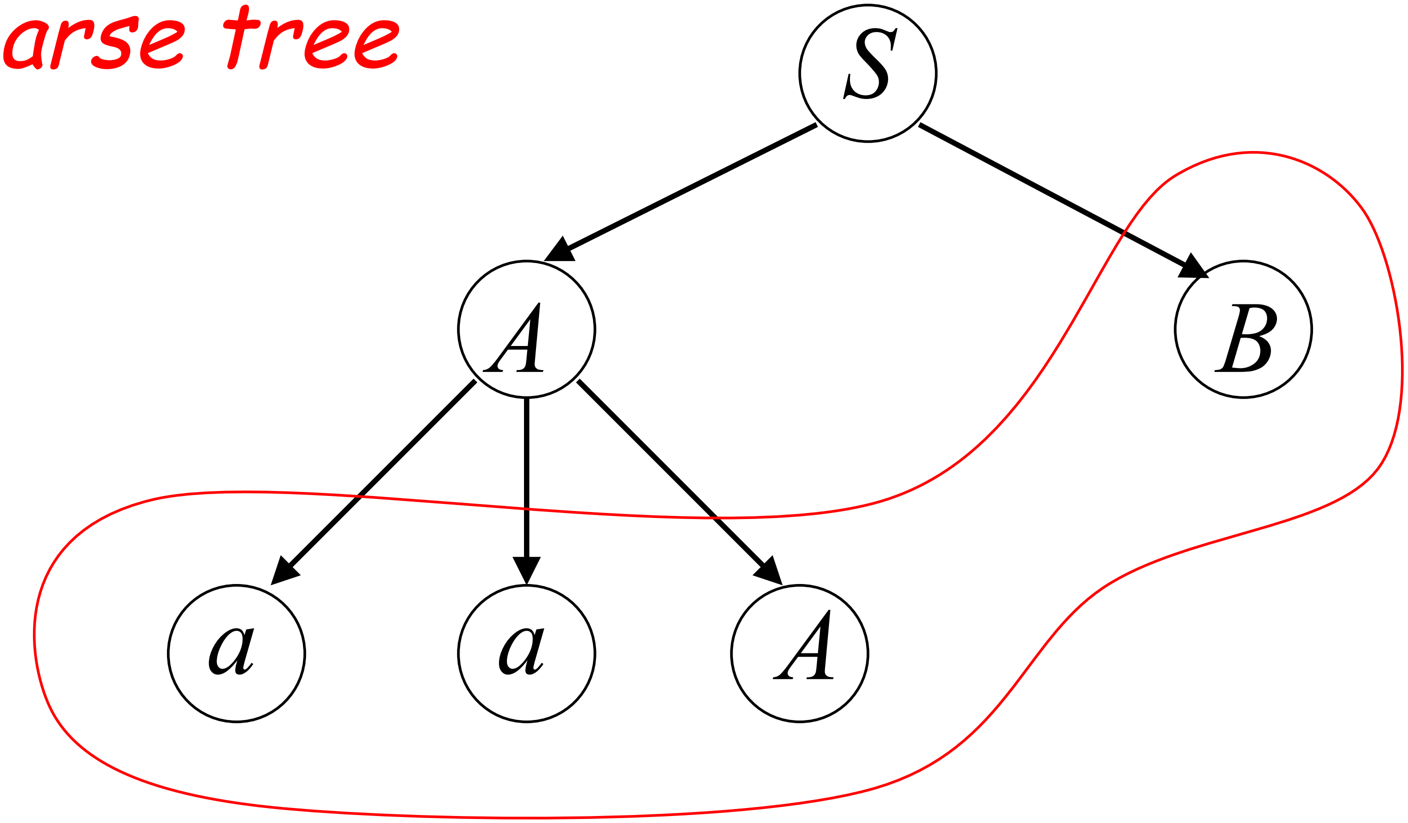
Partial parse tree



$$S \Rightarrow AB \Rightarrow aaAB$$

*sentential form*

*Partial parse tree*



# Sometimes, derivation order doesn't matter

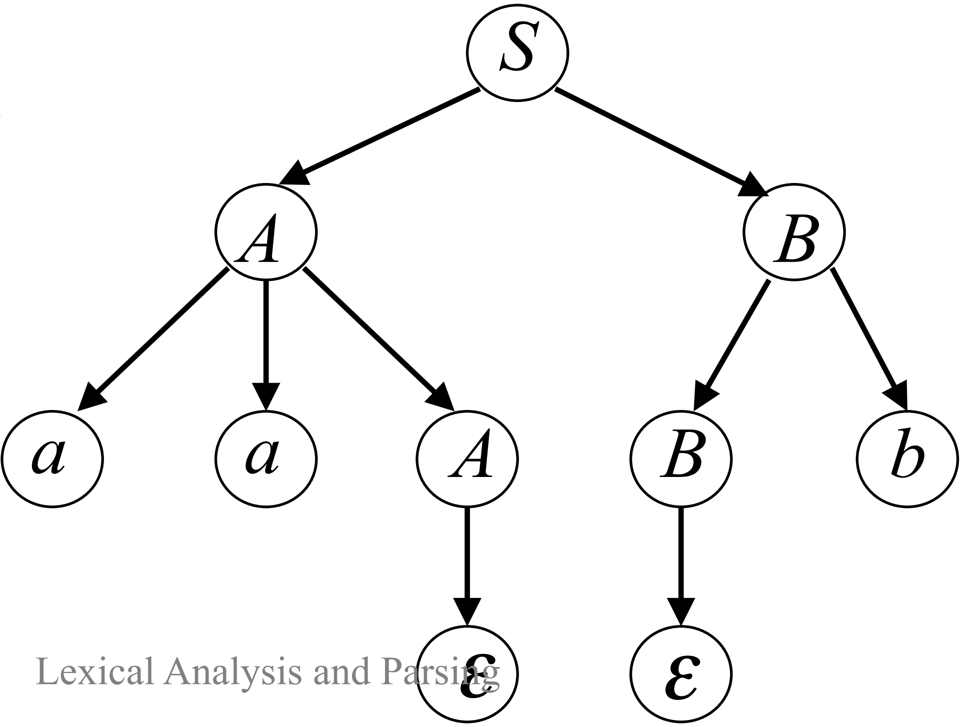
**Leftmost:**

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

**Rightmost:**

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

**Same parse tree**



# How about here?

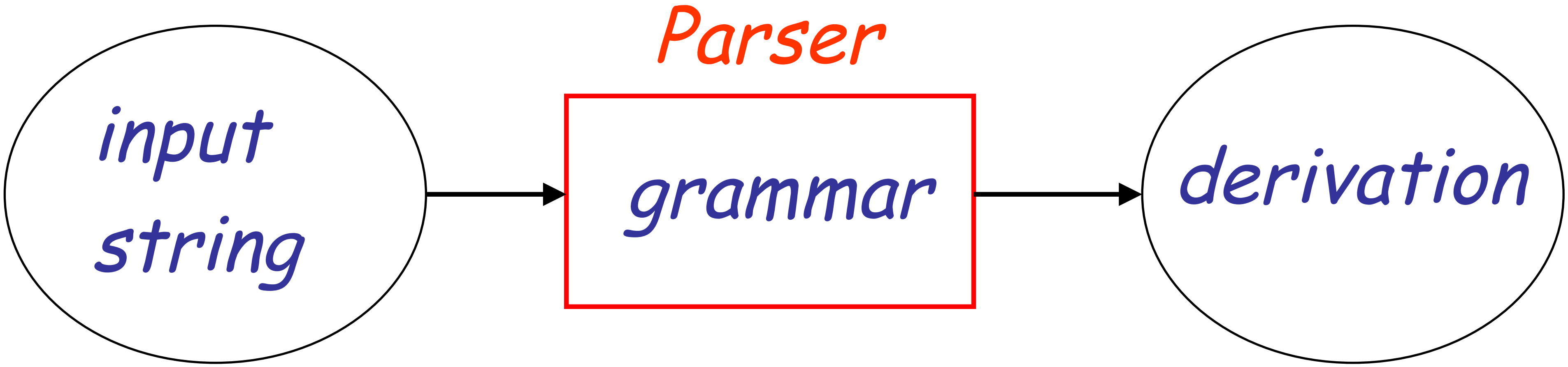
## Grammar

(1) exp --> exp * exp<br>
(2) exp --> exp + exp<br>
(3) exp --> NUM

## String

42 + 7 * 6

*Will be handling this ambiguity later in the semester.*



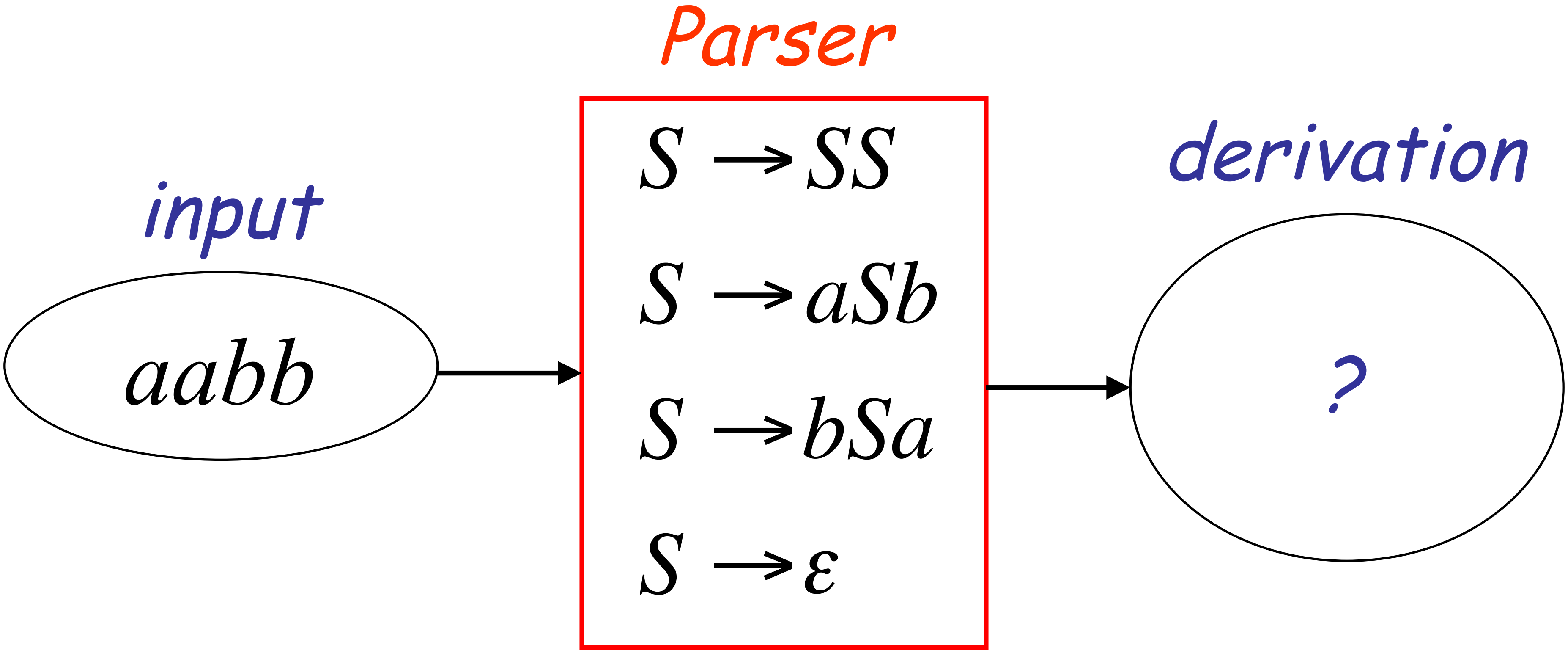
Parser

input string

grammar

derivation

# Example:



input

$aabb$

Parser

$S \rightarrow SS$

$S \rightarrow aSb$

$S \rightarrow bSa$

$S \rightarrow \varepsilon$

derivation

?

# Exhaustive Search

$$S \rightarrow SS \mid aSb \mid bSa \mid \varepsilon$$

Phase 1:

$$S \Rightarrow SS$$

$$S \Rightarrow aSb$$

$$S \Rightarrow bSa$$

$$S \Rightarrow \varepsilon$$

Find derivation of

$$aabb$$

All possible derivations of length 1

$$S \Rightarrow SS$$

$$aabb$$

$$S \Rightarrow aSb$$

$$\xcancel{S \Rightarrow bSa}$$

$$\xcancel{S \Rightarrow \varepsilon}$$

**Phase 2** $S \rightarrow SS \mid aSb \mid bSa \mid \varepsilon$

$aabb$

**Phase 1**

$S \Rightarrow SS$

$S \Rightarrow SS \Rightarrow SSS$

$S \Rightarrow SS \Rightarrow aSbS$

$\cancel{S \Rightarrow SS \Rightarrow bSaS}$

$S \Rightarrow SS \Rightarrow S$

$S \Rightarrow aSb$

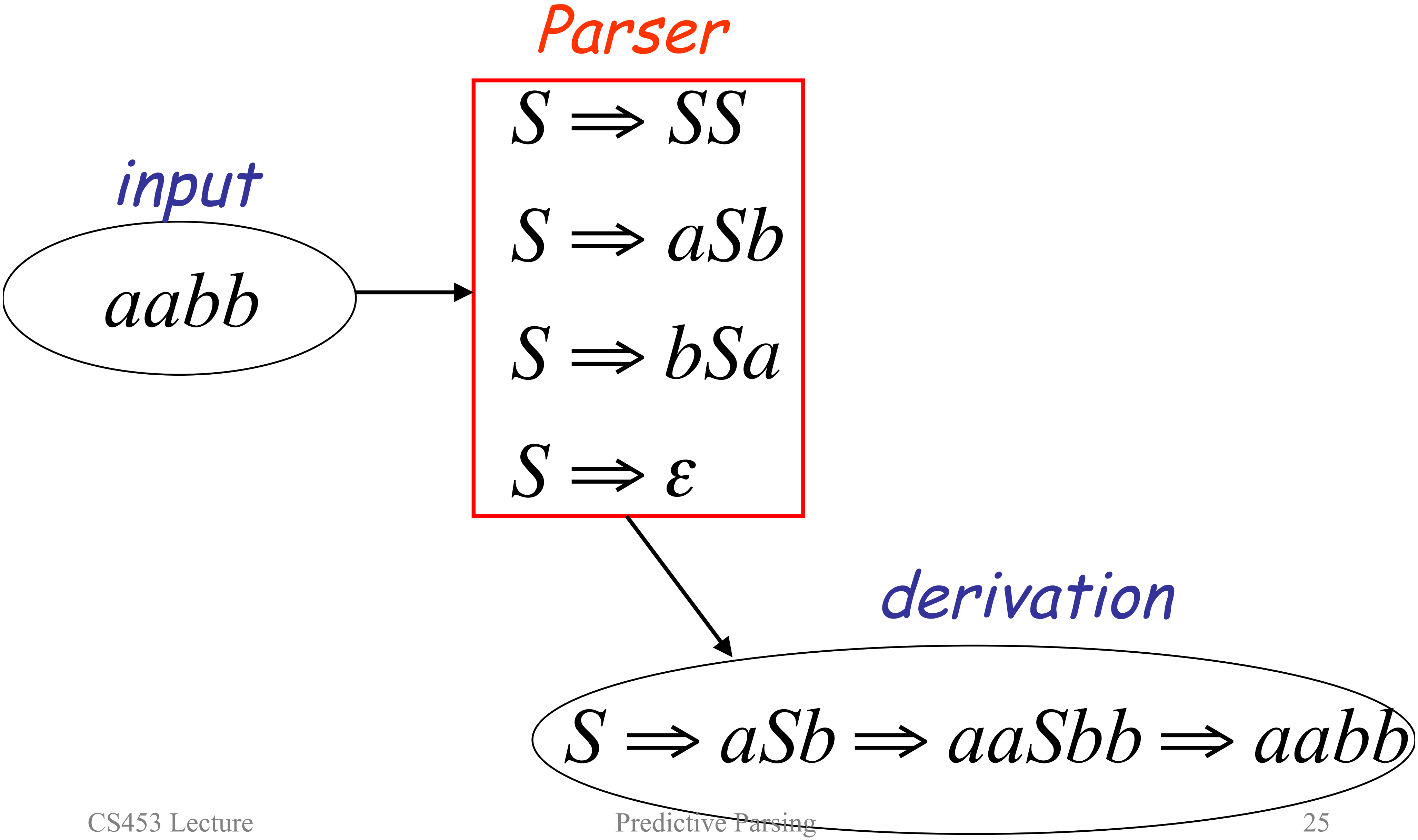
$S \Rightarrow aSb \Rightarrow aSSb$

$S \Rightarrow aSb \Rightarrow aaSbb$

$\cancel{S \Rightarrow aSb \Rightarrow abSab}$

$\cancel{S \Rightarrow aSb \Rightarrow ab}$

# Final result of exhaustive search (top-down parsing)



Parser

$S \Rightarrow SS$

$S \Rightarrow aSb$

$S \Rightarrow bSa$

$S \Rightarrow \varepsilon$

input

$aabb$

derivation

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

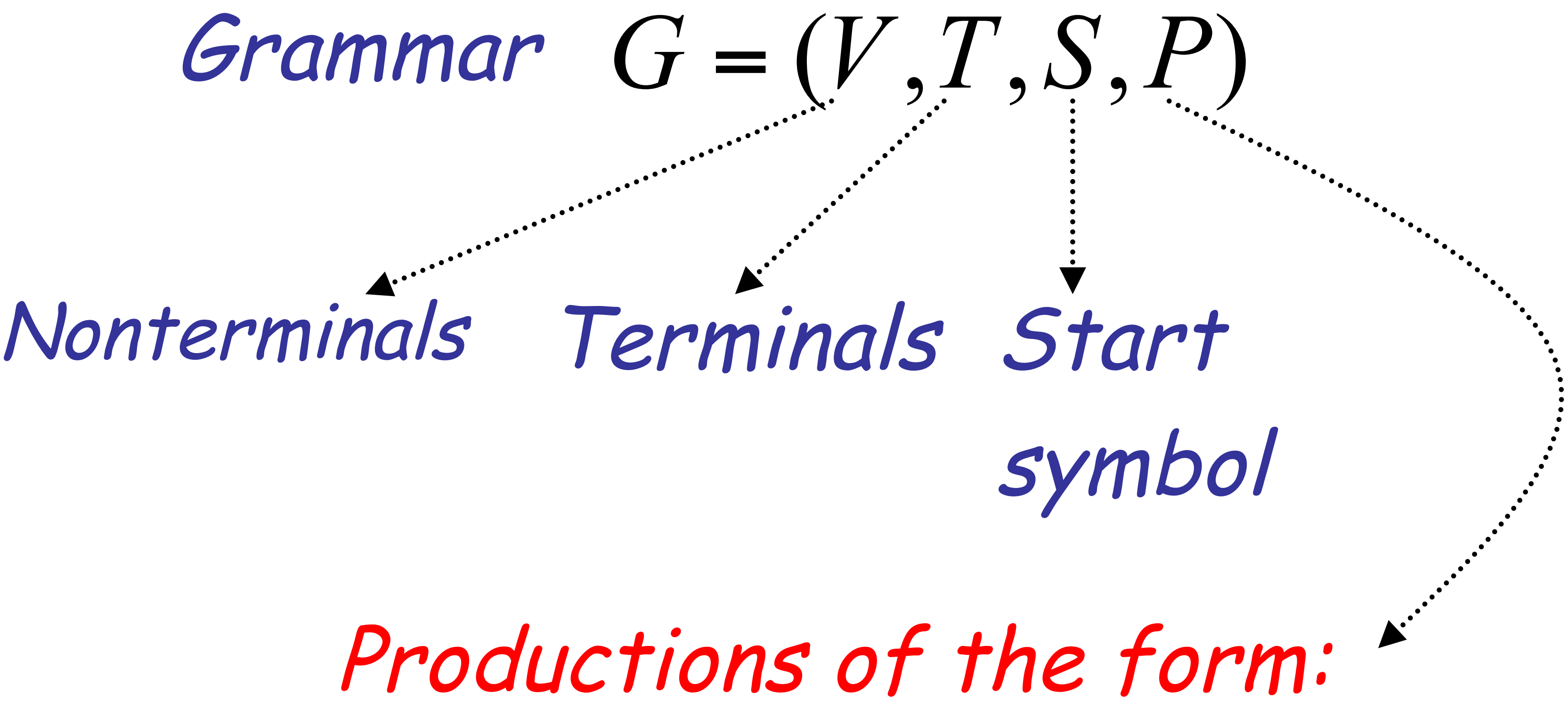
# For general context-free grammars:

The exhaustive search approach is extremely costly: $O(|P|^{|w|})$

There exists a parsing algorithm that parses a string w in time $|w|^3$ for any CFG (Earley parser)

For LL(1) grammars, a simple type of CFGs that we will meet soon, we can use Predictive parsing and parse in $|w|$ time

# Context-Free Grammars



Grammar $G = (V, T, S, P)$

Nonterminals

Terminals

Start symbol

Productions of the form:

$$A \to x$$

Nonterminal

String of symbols, Nonterminals and terminals

# Predictive Parsing

**Predictive parsing,** such as recursive descent parsing, creates the parse tree TOP DOWN, starting at the start symbol, and doing a LEFT-MOST derivation.

**For each non-terminal N** there is a function recognizing the strings that can be produced by N, with one (case) clause for each production.

**Consider:**

```
start      -> stmts EOF
stmts      ->  ε | stmt stmts
stmt       -> ifStmt |  whileStmt | ID = NUM
ifStmt     -> IF id { stmts }
whileStmt  -> WHILE id { stmts }
```

can each production clause be uniquely identified by looking ahead one token? **Let's predictively build the parse tree for**

**if t { while b { x = 6 }} $**

# Example Predictive Parser: Recursive Descent

```
start      -> stmts EOF
stmts      -> ε | stmt stmts
stmt       -> ifStmt |  whileStmt
ifStmt     -> IF id { stmts }
whileStmt  -> WHILE id { stmts }
```

```
void start() { switch(m_lookahead) {
    case IF, WHILE, EOF: stmts(); match(Token.Tag.EOF); break;
    default:    throw new ParseException(…);
}}
void stmts() { switch(m_lookahead) {
    case IF,WHILE:  stmt(); stmts(); break;
    case EOF:       break;
    default:        throw new ParseException(…);
}}
void stmt() { switch(m_lookahead) {
    case IF:    ifStmt();break;
    case WHILE: whileStmt(); break;
    default:    throw new ParseException(…);
}}
void ifStmt() {switch(m_lookahead) {
    case IF: match(id); match(OPENBRACE);
             stmts(); match(CLOSEBRACE); break;
    default: throw new ParseException(…);
}}
```

# Recursive Descent Parsing

Each non-terminal becomes a function

that mimics the RHSs of the productions associated with it

and choses a particular RHS:

an alternative based on a look-ahead symbol

and throws an exception if no alternative applies

# First

**Given a phrase γ of non-terminals and terminals (a rhs of a production), FIRST(γ) is the set of all terminals that can begin a string derived from γ.**

Assume T, F, X, Y, and Z are non-terminals. * is a terminal.

**FIRST(T*F) = ?**

**FIRST(F)= ?**

**FIRST(XYZ) = FIRST(X) ?**

***NO! X could produce ε and then FIRST(Y) comes into play***

***we must keep track of which non terminals are NULLABLE***

# FIRST and Nullable example

```
start      -> stmts EOF
stmts      ->  ε | stmt stmts
stmt       -> ifStmt |  whileStmt | ID = NUM
ifStmt     -> IF id { stmts }
whileStmt  -> WHILE id { stmts }
```

# Follow

**It also turns out to be useful to determine which terminals can directly follow a non terminal X (to decide parsing X is finished).**

**terminal t is in FOLLOW(X) if there is any derivation containing Xt.**

**This can occur if the derivation contains XYZt and Y and Z are nullable**

# FIRST and FOLLOW sets

**NULLABLE**

- X is a nonterminal
- nullable(X) is true if X can derive the empty string

**FIRST**

- FIRST(z) = {z}, where z is a terminal
- FIRST(X) = union of all FIRST( $rhs_i$ ), where X is a nonterminal and
  X -> $rhs_i$ is a production
- FIRST($rhs_i$) = union all of FIRST(sym) on rhs up to and including first nonnullable

**FOLLOW(Y), only relevant when Y is a nonterminal**

- look for Y in rhs of rules (lhs -> rhs) and union all FIRST sets for symbols after Y up to and including first nonnullable
- if all symbols after Y are nullable then also union in FOLLOW(lhs)

# Constructive Definition of nullable, first and follow

**for each terminal t, FIRST(t)={t}**

**Another Transitive Closure algorithm:**

keep doing STEP until nothing changes

Y is a terminal, non-terminal, or epsilon

```
STEP:
for each production X → $Y_1$ $Y_2$ … $Y_k$
 0: if $Y_1$to $Y_k$ nullable, then nullable(X) = true
 for each i from 1 to k, each j from i+1 to k
   1: if $Y_1 \ldots Y_{i-1}$ nullable (or i=1) FIRST(X) += FIRST($Y_i$) //+: union
   2: if $Y_{i+1} \ldots Y_k$ nullable (or i=k) FOLLOW($Y_i$) += FOLLOW(X)
   3: if $Y_{i+1} \ldots Y_{j-1}$ nullable (or i+1=j) FOLLOW($Y_i$) += FIRST($Y_j$)
```

**We can compute nullable, then FIRST, and then FOLLOW**

# Class Exercise

**Compute nullable, FIRST and FOLLOW for**

**Z → d | X Y Z**

**X → a | Y**

**Y → c | ε**

# Constructing the Predictive Parser Table

**A predictive parse table has a row for each non-terminal X, and a column for each input token t. Entries table[X,t] contain productions:**

```
for each X -> gamma
    for each t in FIRST(gamma)
        table[X,t] = X->gamma
    if gamma is nullable
        for each t in FOLLOW(X)
            table[X,t] = X->gamma
```

***Compute the predictive parse table for***

***$Z \rightarrow d \mid X\ Y\ Z$***

***$X \rightarrow a \mid Y$***

***$Y \rightarrow c \mid \varepsilon$***

| | $a$ | $c$ | $d$ |
|---|---|---|---|
| $X$ | $X \rightarrow a$<br>$X \rightarrow Y$ | $X \rightarrow Y$ | $X \rightarrow Y$ |
| $Y$ | $Y \rightarrow \varepsilon$ | $Y \rightarrow \varepsilon$<br>$Y \rightarrow c$ | $Y \rightarrow \varepsilon$ |
| $Z$ | $Z \rightarrow XYZ$ | $Z \rightarrow XYZ$ | $Z \rightarrow XYZ$<br>$Z \rightarrow d$ |

# One more time

**Balanced parentheses grammar 1:**

$$S \rightarrow ( S ) \mid SS \mid \varepsilon$$

1. **Augment the grammar with EOF/$**

2. **Construct Nullable, First and Follow**

3. **Build the predictive parse table, what happens?**

# One more time, but this time with feeling …

**Balanced parentheses grammar 2:**

**$S \rightarrow ( S )S \mid \varepsilon$**

**1. Augment the grammar with EOF/$**

**2. Construct Nullable, First and Follow**

**3. Build the predictive parse table**

**4. Using the predictive parse table, construct the parse tree for**

**( ) ( ( ) ) $**

**and**

**( ) ( ) ( ) $**