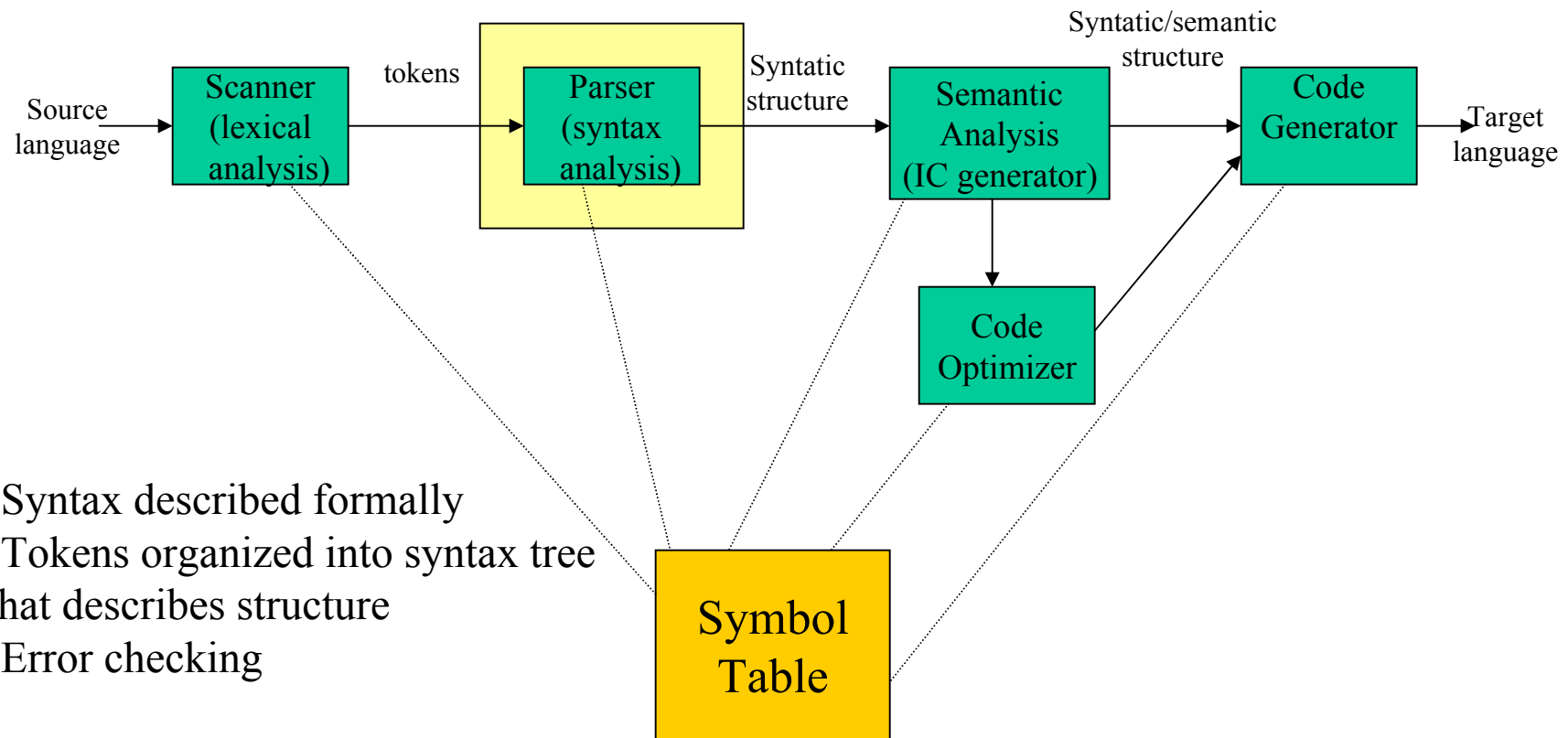


# Lecture 4: LL Parsing

CS 540

George Mason University

# Parsing



- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking

# Top Down (LL) Parsing

$P \rightarrow \textit{begin SS end}$

P

$SS \rightarrow S ; SS$

$SS \rightarrow \epsilon$

$S \rightarrow \textit{simplestmt}$

$S \rightarrow \textit{begin SS end}$

*begin simplestmt ; simplestmt ; end*

# Top Down (LL) Parsing

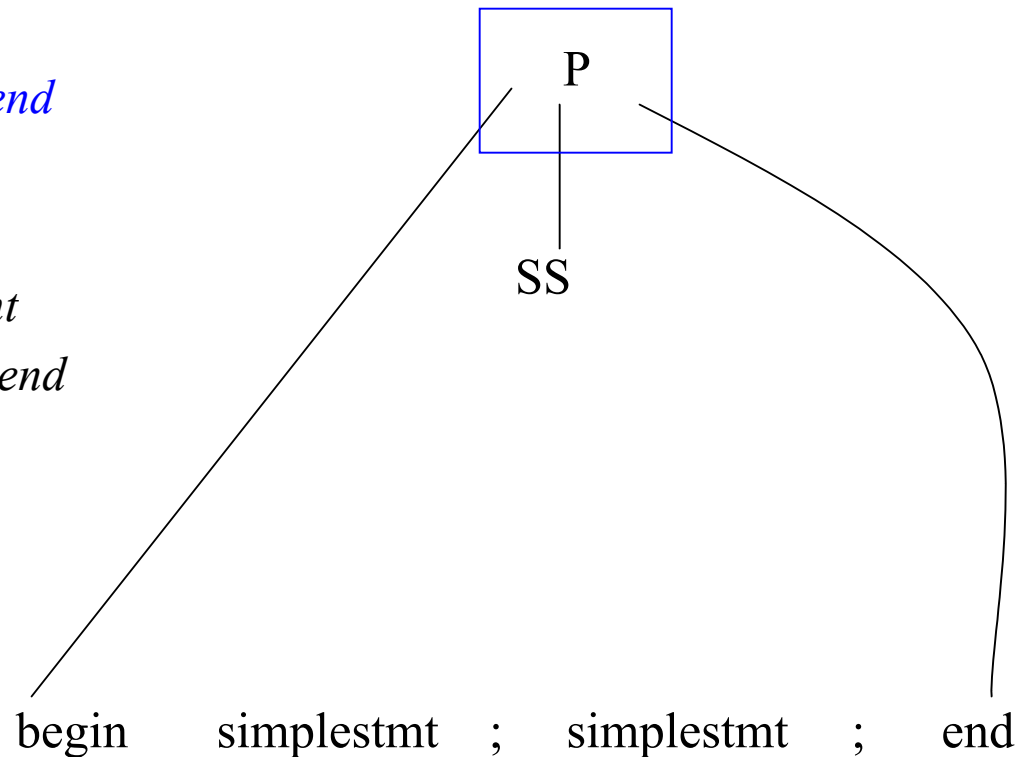
$P \rightarrow \textit{begin SS end}$

$SS \rightarrow S ; SS$

$SS \rightarrow \epsilon$

$S \rightarrow \textit{simplestmt}$

$S \rightarrow \textit{begin SS end}$



# Top Down (LL) Parsing

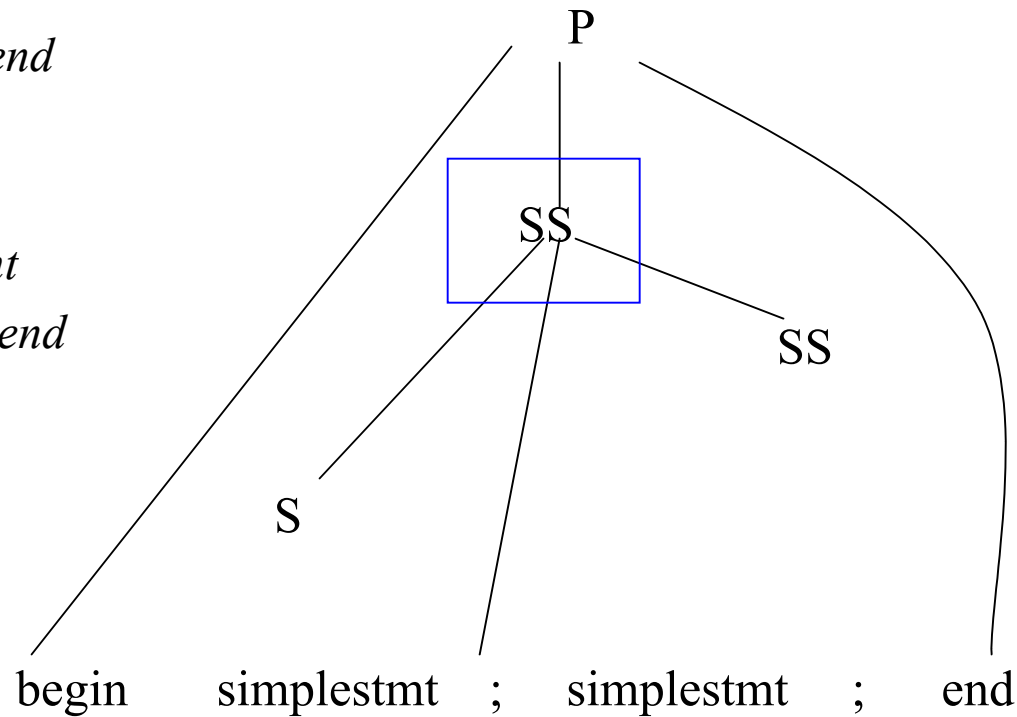
$P \rightarrow \text{begin } SS \text{ end}$

$SS \rightarrow S ; SS$

$SS \rightarrow \epsilon$

$S \rightarrow \text{simplestmt}$

$S \rightarrow \text{begin } SS \text{ end}$



# Top Down (LL) Parsing

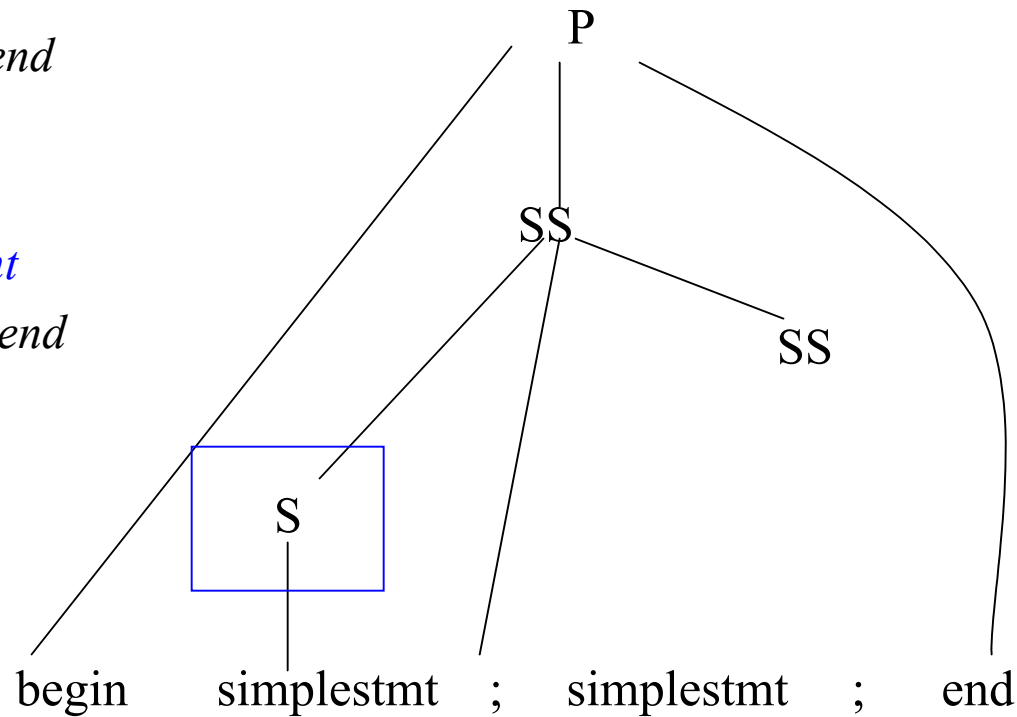
$P \rightarrow \text{begin } SS \text{ end}$

$SS \rightarrow S ; SS$

$SS \rightarrow \epsilon$

$S \rightarrow \text{simplestmt}$

$S \rightarrow \text{begin } SS \text{ end}$





# Top Down (LL) Parsing

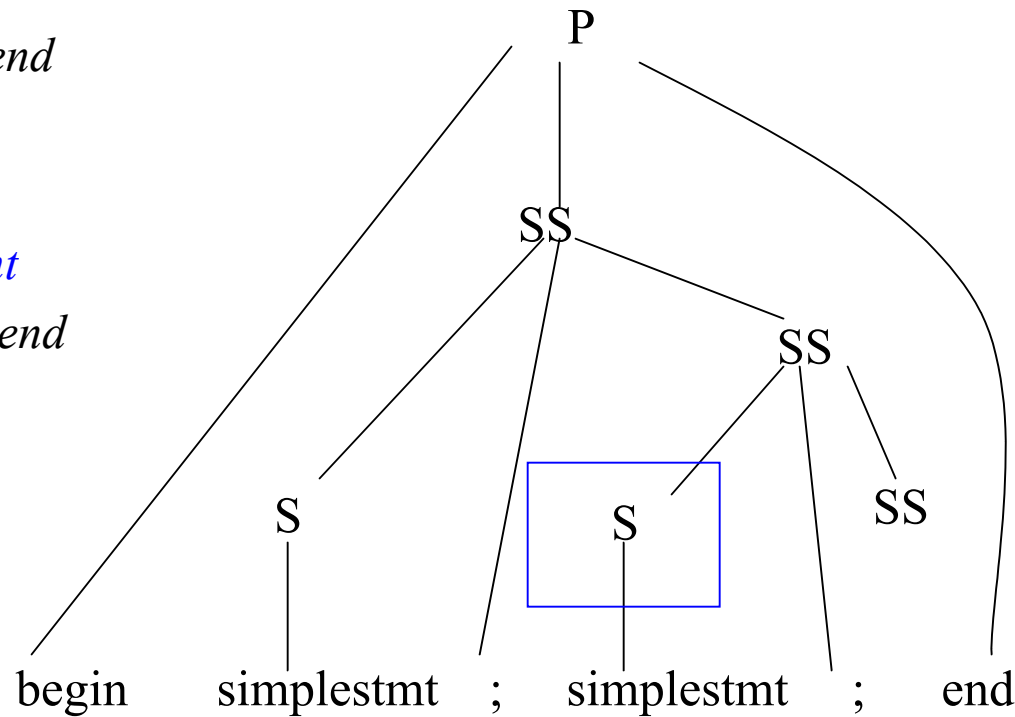
$P \rightarrow \text{begin } SS \text{ end}$

$SS \rightarrow S ; SS$

$SS \rightarrow \epsilon$

$S \rightarrow \text{simplestmt}$

$S \rightarrow \text{begin } SS \text{ end}$



# Top Down (LL) Parsing

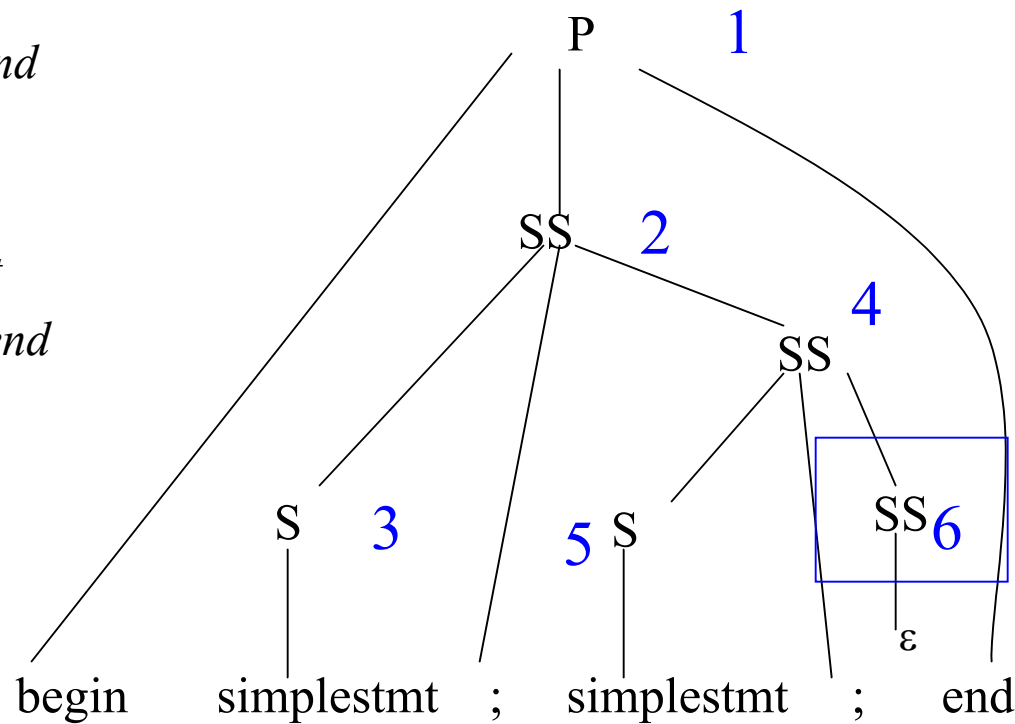
$P \rightarrow \textit{begin SS end}$

$SS \rightarrow S ; SS$

$SS \rightarrow \epsilon$

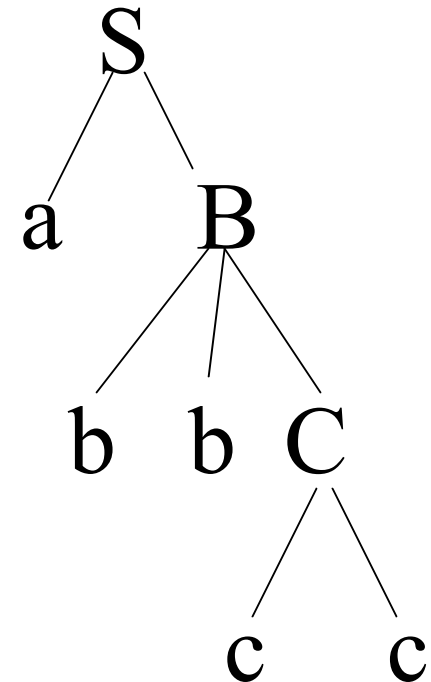
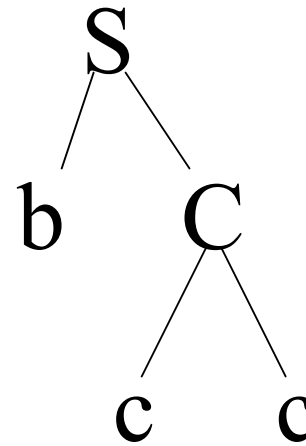
$S \rightarrow \textit{simplestmt}$

$S \rightarrow \textit{begin SS end}$



# Grammar

$S \rightarrow a B$   
 $\quad \quad | b C$   
 $B \rightarrow b b C$   
 $C \rightarrow c c$



Two strings in the language: **abbcc**  
and **bcc**

Can choose between them based on  
the first character of the input.

# LL( $k$ ) parsing

also known as  
the lookahead

- Process input  $k$  symbols at a time.
- Initially, ‘current’ non-terminal is start symbol.
- Algorithm
  - Loop until no more input
    - Given next  $k$  input tokens and ‘current’ non-terminal  $T$ , choose a rule  $R (T \rightarrow \dots)$
    - For each element  $X$  in rule  $R$  from left to right,
      - if  $X$  is a non-terminal, we will need to ‘expand’  $X$
      - else if symbol  $X$  is a terminal, see if next input symbol matches  $X$ ; if so, update from the input
- Typically, we consider **LL(1)**

# Two Approaches

- Recursive Descent parsing
  - Code tailored to the grammar
- Table Driven – predictive parsing
  - Table tailored to the grammar
  - General Algorithm

Both algorithms driven by the tokens coming from the lexer.

# Writing a Recursive Descent Parser

- Generate a procedure for each non-terminal.  
Use next token from `yylex()` (**lookahead**) to choose (PREDICT) which production to ‘mimic’.
  - for non-terminal  $X$ , call procedure  $X()$
  - for terminals  $X$ , call ‘`match(X)`’

Ex:  $B \rightarrow b C D$

```
B() {  
    if (lookahead == 'b')  
        { match('b'); C(); D(); }  
        else ...  
}
```

# Writing a Recursive Descent Parser

Also need the following:

```
match(symbol) {
    if (symbol == lookahead)
        lookahead = yylex()
    else error() }
main() {
    lookahead = yylex();
    S(); /* S is the start symbol */
    if (lookahead == EOF) then accept
    else reject
}
error() { ...
}
```

# Back to grammar

**S()** {

**if (lookahead == a ) { match(a);B(); }**

**else if (lookahead == b) { match(b); C(); }**

**else error(“expecting a or b”);**

**}**

**S → a B**

**S → b C**

**B()** {

**if (lookahead == b)**

**{match(b); match(b); C();}**

**else error();**

**}**

**B → b b C**

**C()** {

**if (lookahead == c)**

**{ match(c) ; match(c) ;}**

**else error();**

**}**

**C → c c**

# Parsing abbcc

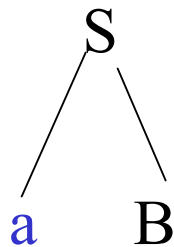
S

Remaining input: **abbcc**

**Call S() from main()**

```
S() {  
    if (lookahead == a ) { match(a);B(); }      S → a B  
    else if (lookahead == b) { match(b); C(); }  S → b C  
    else error(“expecting a or b”);  
}
```

# Parsing abbcc



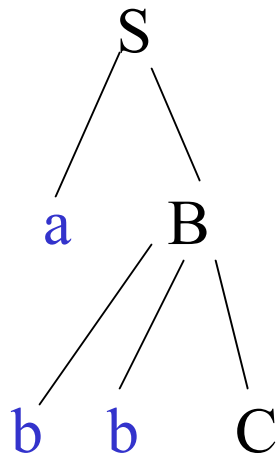
Remaining input: **b**bcc

**Call B() from A():**

```
B() {  
  if (lookahead == b)  
    {match(b); match(b); C();}  
  else error();  
}
```

**B → b b C**

# Parsing abbcc



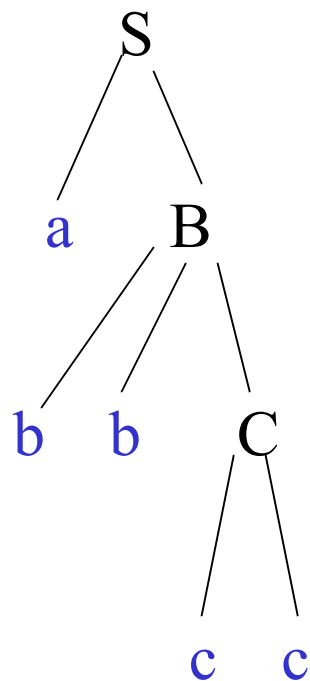
Remaining input: **cc**

**Call C() from B():**

```
C() {  
  if (lookahead == c)  
    { match(c) ; match(c) ;}  
  else error();  
}
```

**C → c c**

# Parsing abbcc



Remaining input:

# How do we find the lookaheads?

- Can compute PREDICT sets from FIRST and FOLLOW for LL(1) parsing:
- $\text{PREDICT}(A \rightarrow \alpha)$ 
  - =  $(\text{FIRST}(\alpha) - \{\epsilon\}) \cup \text{FOLLOW}(A)$       if  $\epsilon$  in  $\text{FIRST}(\alpha)$
  - =  $\text{FIRST}(\alpha)$       if  $\epsilon$  not in  $\text{FIRST}(\alpha)$

NOTE:  $\epsilon$  never in PREDICT sets

For LL(k) grammars, the PREDICT sets for the productions associated with a given non-terminal must be disjoint.

# Example

Production	Predict
$E \rightarrow T E'$	$= \text{FIRST}(T) = \{ (, \text{id} \}$
$E' \rightarrow + T E'$	$\{ + \}$
$E' \rightarrow \varepsilon$	$= \text{FOLLOW}(E') = \{ \$, ) \}$
$T \rightarrow F T'$	$= \text{FIRST}(F) = \{ (, \text{id} \}$
$T' \rightarrow * F T'$	$\{ * \}$
$T' \rightarrow \varepsilon$	$= \text{FOLLOW}(T') = \{ +, \$, ) \}$
$F \rightarrow \text{id}$	$\{ \text{id} \}$
$F \rightarrow ( E )$	$\{ ( \}$

$\text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(T) = \{ (, \text{id} \}$

$\text{FIRST}(E) = \{ (, \text{id} \}$

$\text{FIRST}(T') = \{ *, \varepsilon \}$

$\text{FIRST}(E') = \{ +, \varepsilon \}$

$\text{FOLLOW}(E) = \{ \$, ) \}$

$\text{FOLLOW}(E') = \{ \$, ) \}$

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

$\text{FOLLOW}(T') = \{ +, \$, ) \}$

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

*Assume E is the start symbol*

```

E() {
  if (lookahead in {(,id } ) { T(); E_prime(); }      E → T E'
  else error("E expecting ( or identifier");
}

```

```

E_prime() {
  if (lookahead in {+}) {match(+); T(); E_prime();}  E' → + T E'
  else if (lookahead in {),end_of_file}) return;    E' → ε
  else error("E_prime expecting +, ) or end of file");
}

```

```

T() {
  if (lookahead in {(,id}) { F(); T_prime(); }      T → F T'
  else error("T expecting ( or identifier");
}

```

```

T_prime() {
  if (lookahead in {*}) {match(*); F(); T_prime();}   T' → * F T'
  else if (lookahead in {+,},end_of_file}) return;    T' → ε
  else error("T_prime expecting *, ) or end of file"); }

```

```

F() {
  if (lookahead in {id}) match(id);                   F → id
  else if (lookahead in {(} ) { match( ( ); E(); match ( ) ); } F → ( E )
  else error("F expecting ( or identifier");}

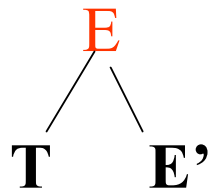
```

# Parsing $a + b * c$

E

Remaining input:  $a+b*c$

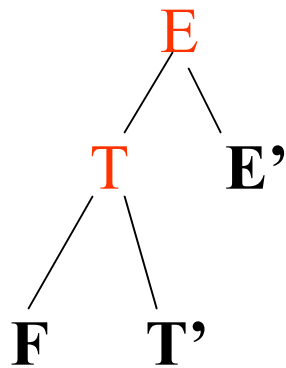
# Parsing $a + b * c$



Remaining input:  $a + b * c$

```
E() {  
  if (lookahead in { (, id } ) { T(); E_prime(); }  
  else error("E expecting ( or identifier");  
}
```

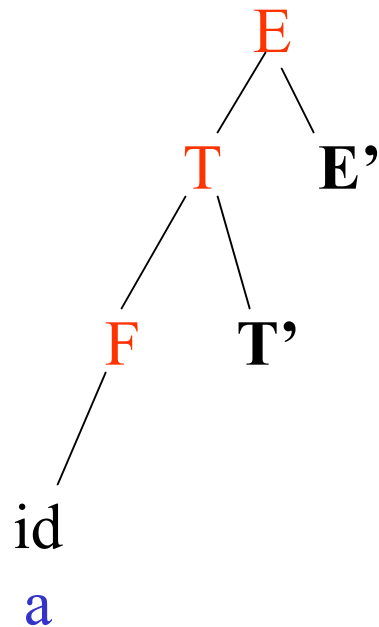
# Parsing $a + b * c$



Remaining input:  $a + b * c$

```
T() {  
  if (lookahead in {(,id } ) { F(); T_prime(); }  
  else error("T expecting ( or identifier");  
}
```

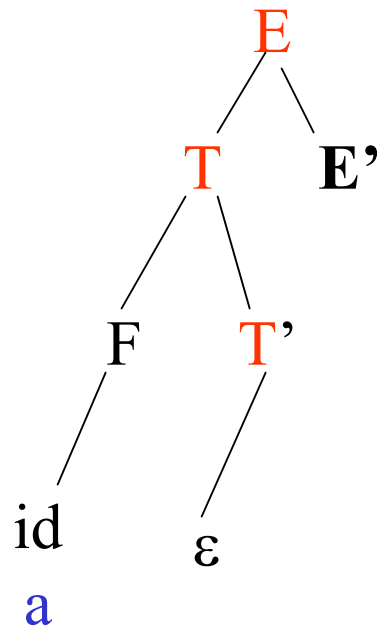
# Parsing $a + b * c$



Remaining input:  $+b*c$

```
F() {  
  if (lookahead in {id } ) match(id)  
  else if (lookahead in { ( } {  
    match( ( ); E(); match( ) );  
  }  
  else error("F expecting ( or identifier");  
}
```

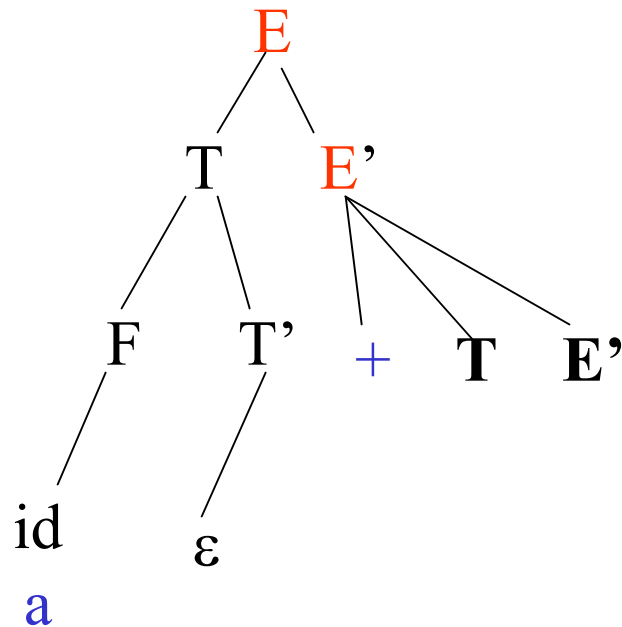
# Parsing $a + b * c$



Remaining input:  $+b*c$

```
T_prime() {  
  if (lookahead in { * }) { match( * ); F(); T_prime(); }  
  else if (lookahead in { +, }, end_of_file ) return;  
  else error( "T_prime expecting *, ) or end of file" );  
}
```

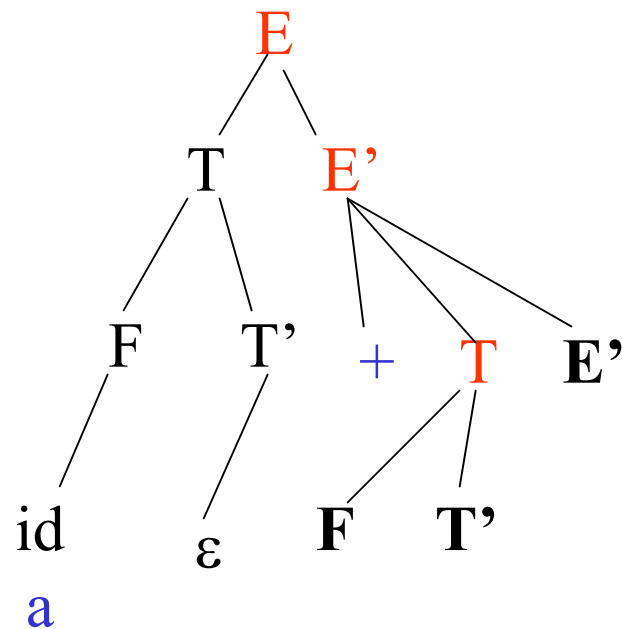
# Parsing $a + b * c$



Remaining input:  $b * c$

```
E_prime() {  
  if (lookahead in {+})  
    {match(+); T(); E_prime();}  
  else if (lookahead in {},end_of_file})  
    return;  
  else  
    error("E_prime expecting *, ) or end of file");  
}
```

# Parsing $a + b * c$

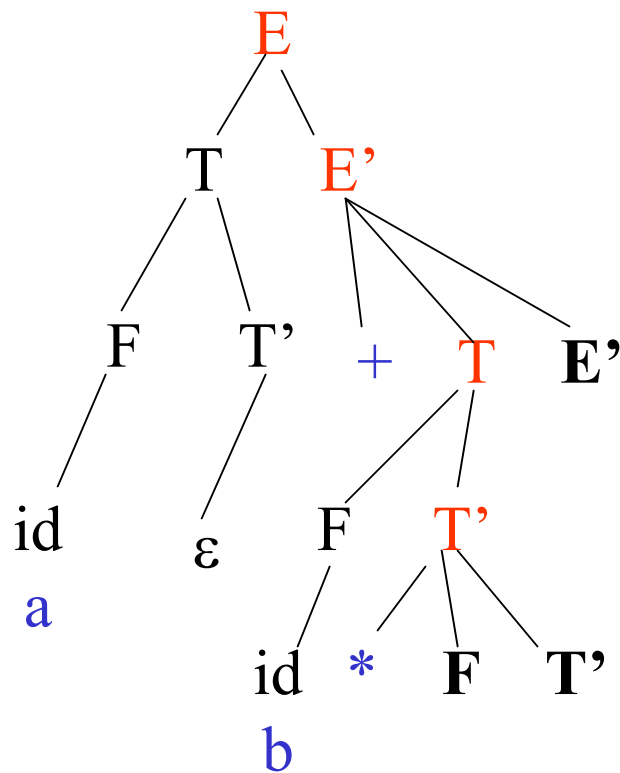


Remaining input:  **$b * c$**

```
T() {  
  if (lookahead in { (, id } ) { F(); T_prime(); }  
  else error("T expecting ( or identifier");  
}
```



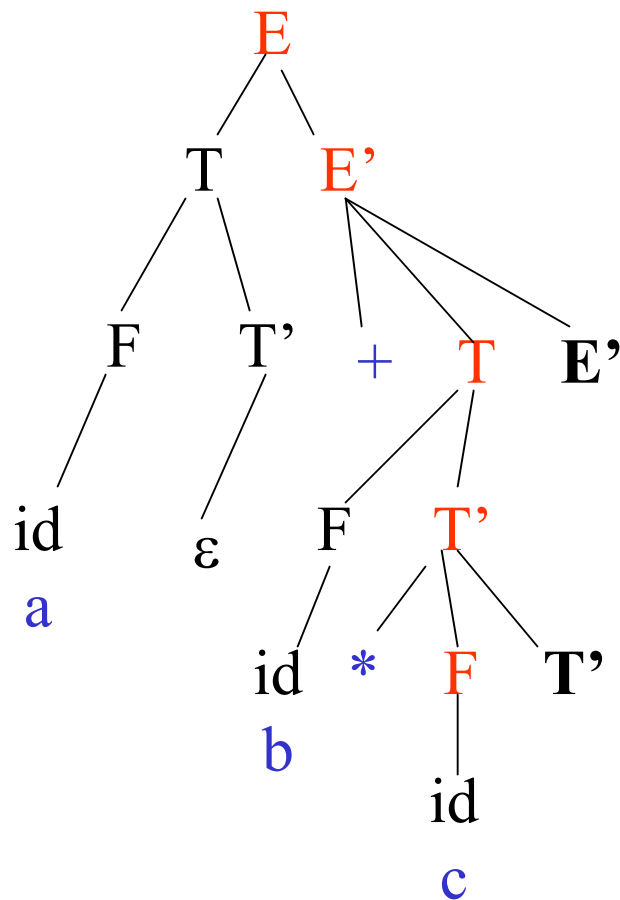
# Parsing $a + b * c$



Remaining input: **c**

```
T_prime() {
  if (lookahead in {*})
    {match(*); F(); T_prime();}
  else if (lookahead in {+,),end_of_file})
    return;
  else
    error("T_prime expecting *, ) or end of file");
}
```

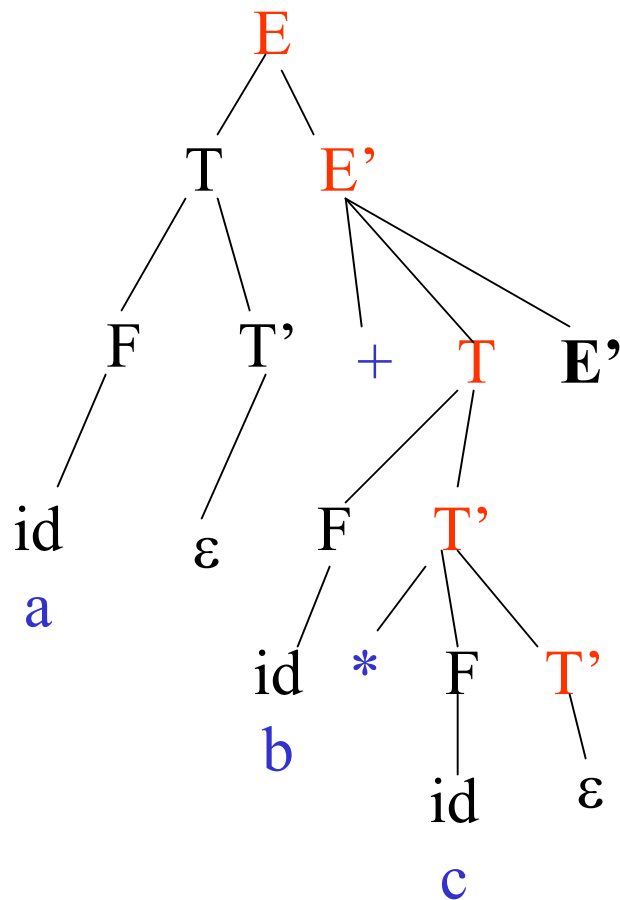
# Parsing $a + b * c$



Remaining input:

```
F() {
  if (lookahead in {id } ) match(id)
  else if (lookahead in { ( } {
    match( ( ); E(); match( ) ); }
  else error("F expecting ( or identifier");
}
```

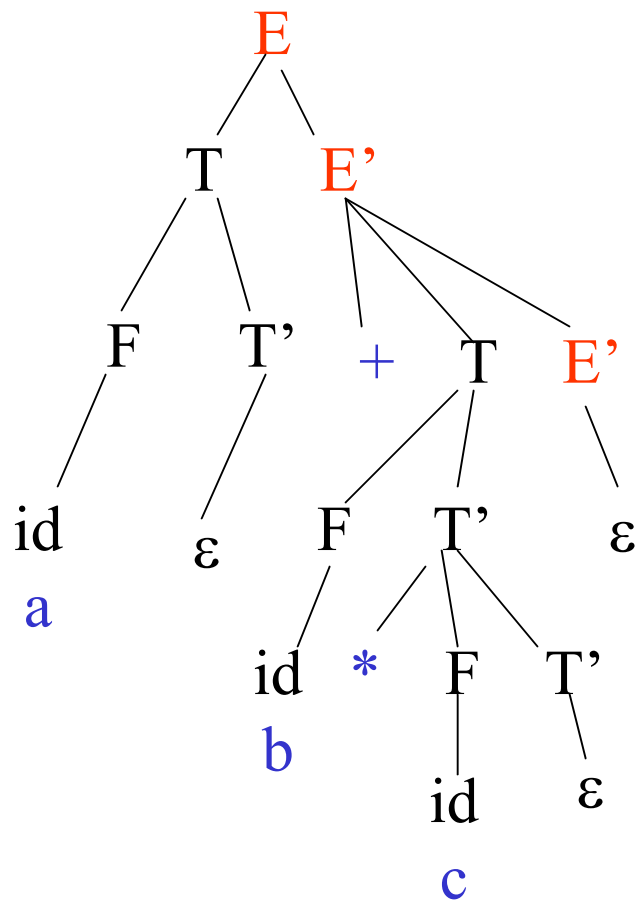
# Parsing $a + b * c$



Remaining input:

```
T_prime() {
  if (lookahead in {*})
    {match(*); F(); T_prime();}
  else if (lookahead in {+,),end_of_file})
    return;
  else
    error("T_prime expecting *, ) or end of file");
}
```

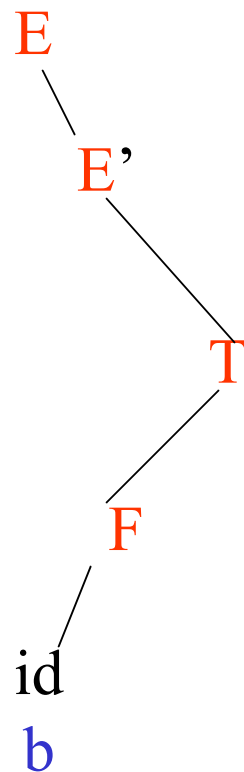
# Parsing $a + b * c$



Remaining input:

```
E_prime() {
  if (lookahead in {+})
    {match(+); T(); E_prime();}
  else if (lookahead in {},end_of_file)
    return;
  else
    error("E_prime expecting *, ) or end of file");
}
```

# Stacks in Recursive Descent Parsing



- Runtime stack
- Procedure activations correspond to a path in parse tree from root to some interior node

# Two Approaches

- Recursive Descent parsing
  - Code tailored to the grammar
- Table Driven – predictive parsing
  - Table tailored to the grammar
  - General Algorithm

Both algorithms driven by the tokens coming from the lexer.

# LL(1) Predictive Parse Tables

An LL(1) Parse table is a mapping T:

$$V_n \times V_t \rightarrow \text{production } P \text{ or error}$$

1. For all productions  $A \rightarrow \alpha$  do

For each terminal  $t$  in  $\text{Predict}(A \rightarrow \alpha)$ ,

$$T[A][t] = A \rightarrow \alpha$$

2. Every undefined table entry is an error.

# Using LL(1) Parse Tables

## ALGORITHM

INPUT: token sequence to be parsed,  
followed by '\$' (end of file)

## DATA STRUCTURES:

- Parse stack: Initialized by pushing '\$' and then pushing the start symbol
- Parse table T

# Algorithm: Predictive Parsing

```
push($); push(start_symbol);
```

```
lookahead = yylex()
```

```
repeat
```

```
  X = pop(stack)
```

```
  if X is a terminal symbol or $ then
```

```
    if X = lookahead then
```

```
      lookahead = yylex()
```

```
    else error()
```

```
  else /* X is non-terminal */
```

```
    if  $T[X][\text{lookahead}] = X \rightarrow Y_1 Y_2 \dots Y_m$ 
```

```
      push( $Y_m$ ) ... push( $Y_1$ )
```

```
    else error()
```

```
until X = $ token
```

similar to 'match'



similar to 'mimic'



# Example

NT/T	+	*	(	)	ID	\$
E						
E'						
T						
T'						
F						

Production	Predict
1: $E \rightarrow T E'$	{(,id}
2: $E' \rightarrow + T E'$	{+}
3: $E' \rightarrow \epsilon$	{\$,)}
4: $T \rightarrow F T'$	{(,id}
5: $T' \rightarrow * F T'$	{*}
6: $T' \rightarrow \epsilon$	{+,\$,)}
7: $F \rightarrow id$	{id}
8: $F \rightarrow ( E )$	{(}

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Production	Predict
1: $E \rightarrow T E'$	{(,id}
2: $E' \rightarrow + T E'$	{+}
3: $E' \rightarrow \epsilon$	{\$,)} }
4: $T \rightarrow F T'$	{(,id}
5: $T' \rightarrow * F T'$	{*}
6: $T' \rightarrow \epsilon$	{+,\$,)} }
7: $F \rightarrow \text{id}$	{id}
8: $F \rightarrow ( E )$	{(}





NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	E → T E'
\$E'T	a+b*c\$	T → F T'
\$E'T'F	a+b*c\$	F → id

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow id$
\$E'T'id	a+b*c\$	match

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow \text{id}$
\$E'T'id	a+b*c\$	match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow \text{id}$
\$E'T'id	a+b*c\$	match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$
\$E'	+b*c\$	$E' \rightarrow + T E'$

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow id$
\$E'T'id	a+b*c\$	match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$
\$E'	+b*c\$	$E' \rightarrow + T E'$
\$E' T +	+b*c\$	match

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow \text{id}$
\$E'T'id	a+b*c\$	match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$
\$E'	+b*c\$	$E' \rightarrow + T E'$
\$E' T +	+b*c\$	match
\$E' T	b*c\$	$T \rightarrow F T'$

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow id$
\$E'T'id	a+b*c\$	match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$
\$E'	+b*c\$	$E' \rightarrow + T E'$
\$E' T +	+b*c\$	match
\$E' T	b*c\$	$T \rightarrow F T'$
\$E'T'F	b*c\$	$F \rightarrow id$

NT/T	+	*	(	)	ID	\$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			8		7	

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T	a+b*c\$	$T \rightarrow F T'$
\$E'T'F	a+b*c\$	$F \rightarrow id$
\$E'T'id	a+b*c\$	match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$
\$E'	+b*c\$	$E' \rightarrow + T E'$
\$E' T +	+b*c\$	match
\$E' T	b*c\$	$T \rightarrow F T'$
\$E'T'F	b*c\$	$F \rightarrow id$
\$E'T id	b*c\$	match

# Parsing $a + b * c$

Stack	Input	Action
\$E	a+b*c\$	$E \rightarrow T E'$
\$E'T		$T \rightarrow F T'$
\$E'T'F		$F \rightarrow \text{id}$
\$E'T'id		match
\$E'T'	+b*c\$	$T' \rightarrow \epsilon$
\$E'		$E' \rightarrow + T E'$
\$E'T+		match
\$E'T	b*c\$	$T \rightarrow F T'$

Stack	Input	Action
\$E'T'F		$F \rightarrow \text{id}$
\$E'T'id		match
\$E'T'	*c\$	$T' \rightarrow * F T'$
\$E'T'F*		match
\$E'T'F	c\$	$F \rightarrow \text{id}$
\$E'T'id		match
\$E'T'	\$	$T' \rightarrow \epsilon$
\$E'		$E' \rightarrow \epsilon$
\$		accept

# Stacks in Predictive Parsing

- Algorithm data structure
- Hold terminals and non-terminals from the grammar
  - terminals – still need to be matched from the input
  - non-terminals – still need to be expanded

# Making a grammar LL(1)

- Not all context free languages have LL(1) grammars
- Can show a grammar is not LL(1) by looking at the predict sets
  - For LL(1) grammars, the PREDICT sets for a given non-terminal will be disjoint.

# Example

Production	Predict
$E \rightarrow E + T$	$= \text{FIRST}(E) = \{(, \text{id}\}$
$E \rightarrow T$	$= \text{FIRST}(T) = \{(, \text{id}\}$
$T \rightarrow T * F$	$= \text{FIRST}(T) = \{(, \text{id}\}$
$T \rightarrow F$	$= \text{FIRST}(F) = \{(, \text{id}\}$
$F \rightarrow \text{id}$	$= \{\text{id}\}$
$F \rightarrow ( E )$	$= \{( \}$

- $\text{FIRST}(F) = \{(, \text{id}\}$
- $\text{FIRST}(T) = \{(, \text{id}\}$
- $\text{FIRST}(E) = \{(, \text{id}\}$
- $\text{FIRST}(T') = \{*, \epsilon\}$
- $\text{FIRST}(E') = \{+, \epsilon\}$
- $\text{FOLLOW}(E) = \{\$, \}$
- $\text{FOLLOW}(E') = \{\$, \}$
- $\text{FOLLOW}(T) = \{+\$, \}$
- $\text{FOLLOW}(T') = \{+\$, \}$
- $\text{FOLLOW}(F) = \{*, +, \$, \}$

Two problems: E and T

# Making a non-LL(1) grammar LL(1)

- Eliminate common prefixes

Ex:  $A \rightarrow B a C D \mid B a C E$

- Transform left recursion to right recursion

Ex:  $E \rightarrow E + T \mid T$

# Eliminate Common Prefixes

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

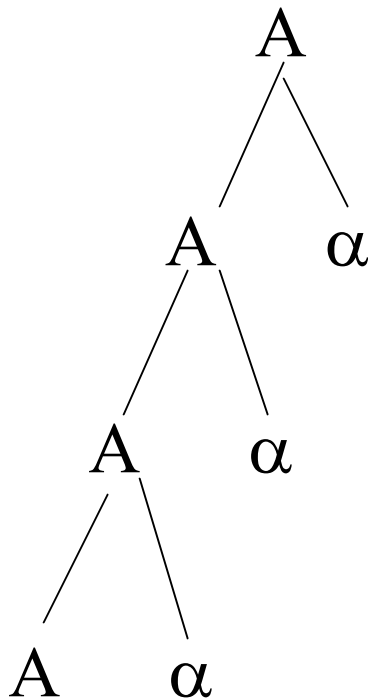
Can become:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta \mid \delta$$

Doesn't always remove the problem. *Why?*

# Why is left recursion a problem?



# Remove Left Recursion

$$A \rightarrow A \alpha_1 \mid A \alpha_2 \mid \dots \mid \beta_1 \mid \beta_2 \mid \dots$$

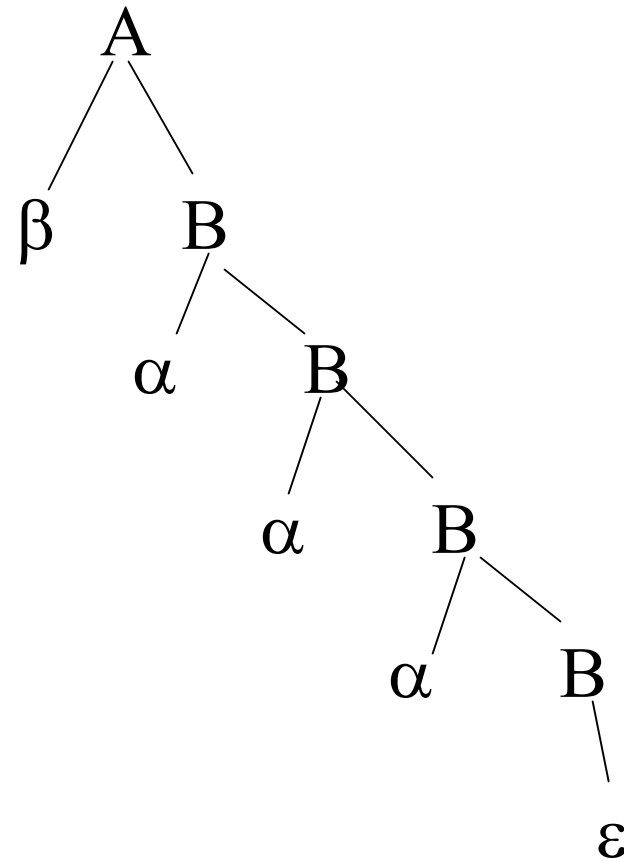
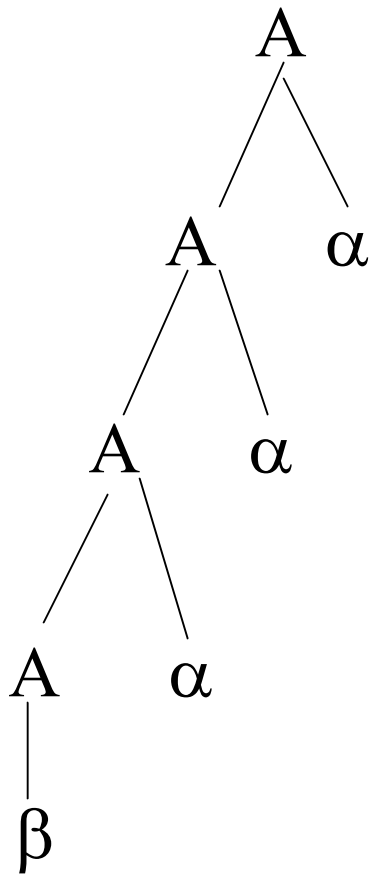
becomes

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \varepsilon$$

The left recursion becomes right recursion

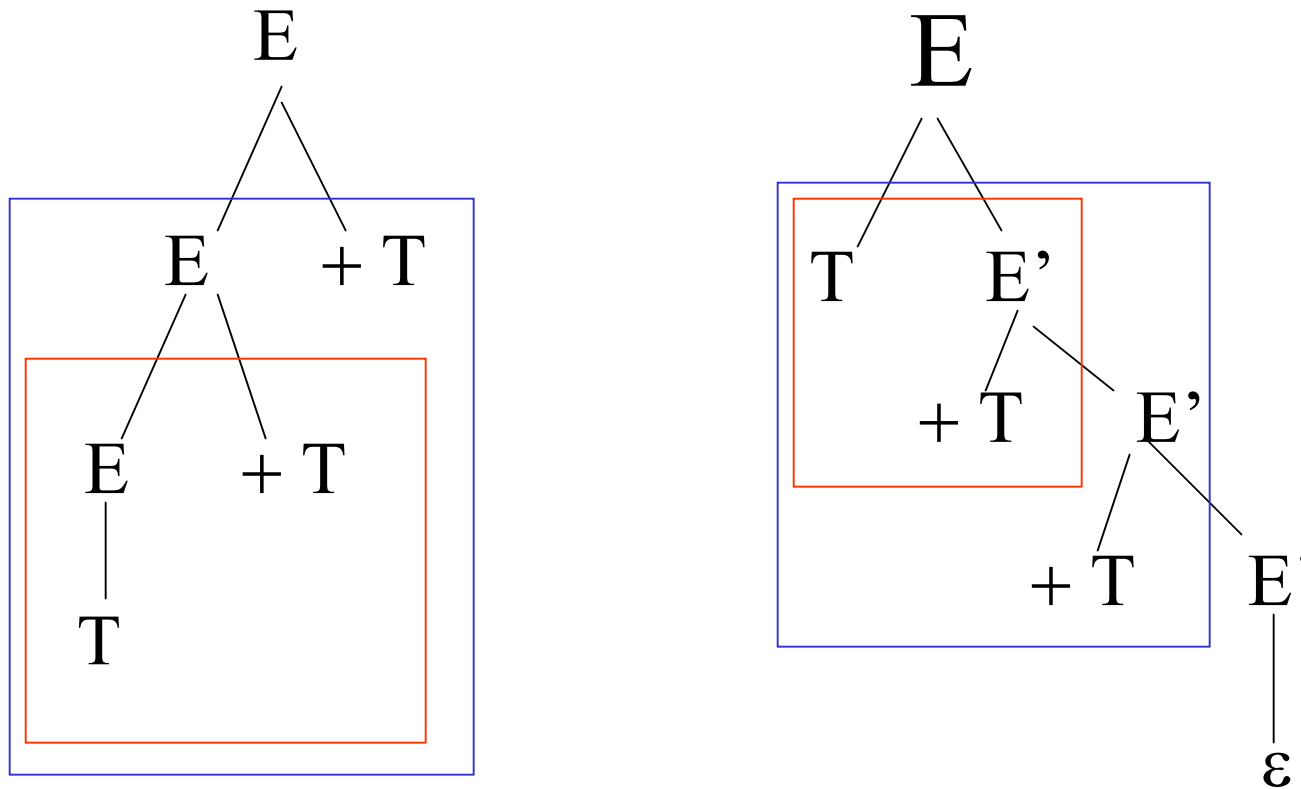
$A \rightarrow A \alpha \mid \beta$  becomes  $A \rightarrow \beta B, B \rightarrow \alpha B \mid \epsilon$



# Expression Grammar

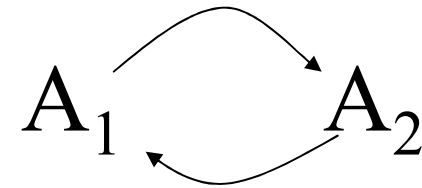
- $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow \text{id} \mid ( E )$  NOT LL(1)
- Eliminate left recursion:  
 $E \rightarrow T E', \quad E' \rightarrow + T E' \mid \varepsilon$   
 $T \rightarrow F T', \quad T' \rightarrow * F T' \mid \varepsilon$   
 $F \rightarrow \text{id} \mid ( E )$

$E \rightarrow E + T \mid T$  becomes  $E \rightarrow T E', E' \rightarrow + T E' \mid \epsilon$



# Non-Immediate Left Recursion

- Ex:  $A_1 \rightarrow A_2 a \mid b$   
 $A_2 \rightarrow A_1 c \mid A_2 d$



- **Convert to immediate left recursion**
  - Substitute  $A_1$  in second set of productions by  $A_1$ 's definition:

$$A_1 \rightarrow A_2 a \mid b$$

$$A_2 \rightarrow A_2 a c \mid b c \mid A_2 d$$

- Eliminate recursion:

$$A_1 \rightarrow A_2 a \mid b$$

$$A_2 \rightarrow b c A_3$$

$$A_3 \rightarrow a c A_3 \mid d A_3 \mid \varepsilon$$

# Example

- $A \rightarrow B c \mid d$

- $B \rightarrow C f \mid B f$

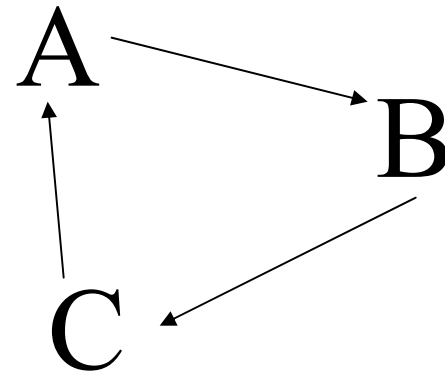
- $C \rightarrow A e \mid g$

- Rewrite: replace C in B

- $B \rightarrow A e f \mid g f \mid B f$

- Rewrite: replace A in B

- $B \rightarrow B c e f \mid d e f \mid g f \mid B f$



- Now grammar is:

$$A \rightarrow B c \mid d$$

$$B \rightarrow B c e f \mid d e f \mid g f \mid B f$$

$$C \rightarrow A e \mid g$$

- Get rid of left recursion (and C if A is start)

$$A \rightarrow B c \mid d$$

$$B \rightarrow d e f B' \mid g f B'$$

$$B' \rightarrow c e f B' \mid f B' \mid \varepsilon$$

# Error Recovery in LL parsing

- Simple option: When see an error, print a message and halt
- “Real” error recovery
  - Insert “expected” token and continue – can have a problem with termination
  - Deleting tokens – for an error for non-terminal F, keep deleting tokens until see a token in follow(F).

For example:

```
E() {  
  if (lookahead in { (, id } ) { T(); E_prime(); }      E → T E'  
  else { printf("E expecting ( or identifier");      Follow(E) = $ )  
    while (lookahead != ) or $) lookahead = yylex();  
  }  
}
```

# Real-World Compilers

<http://cs.gmu.edu/~white/CS540/parser.cpp>

// CParser::ParseSourceModule is the “main”