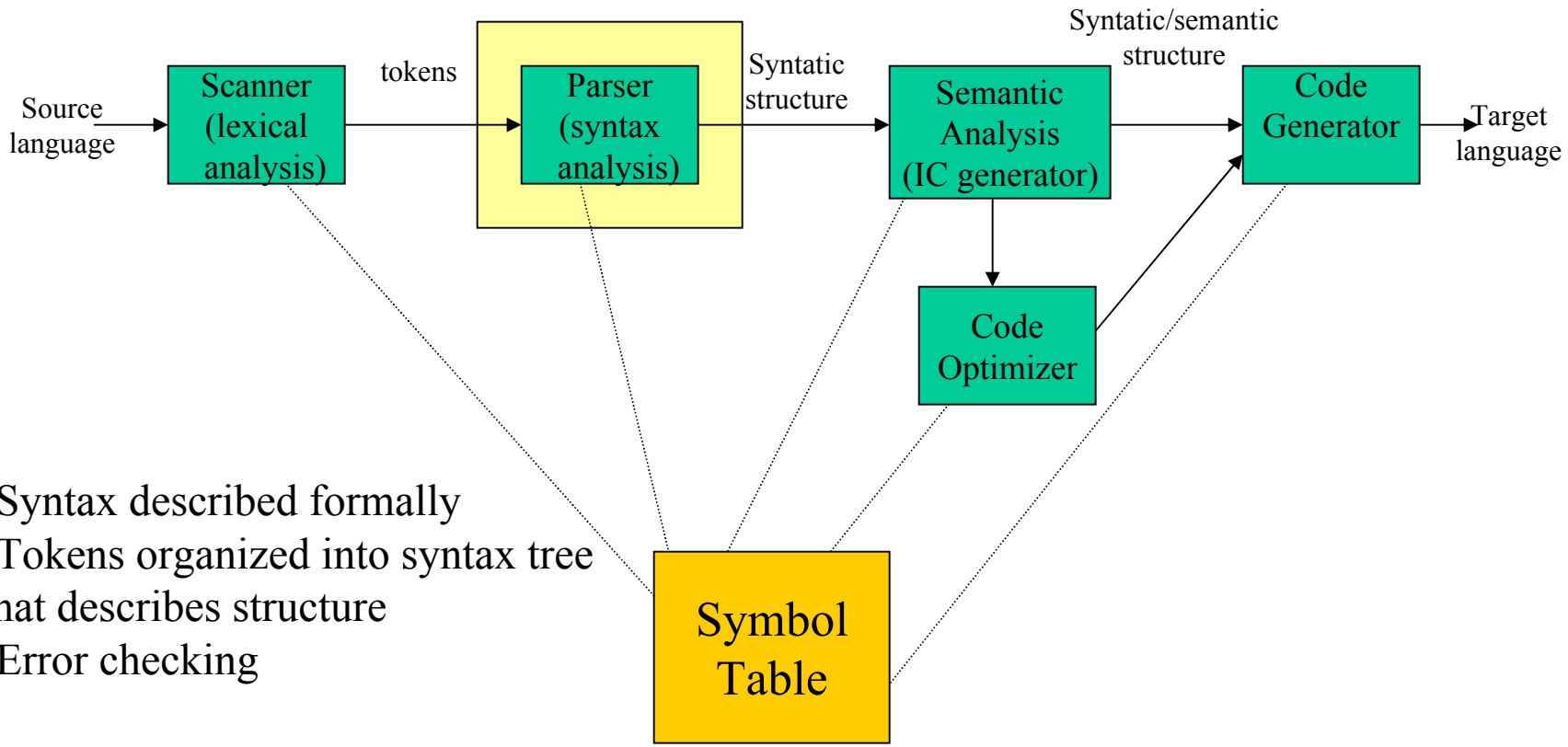
Lecture 4: LL Parsing

CS 540

George Mason University

Parsing



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

Top Down (LL) Parsing

P→ *begin* SS *end*

SS → S *;* SS

SS → ε

S → *simplestmt*

S → *begin* SS *end*

P

begin simplestmt ; simplestmt ; end

Top Down (LL) Parsing

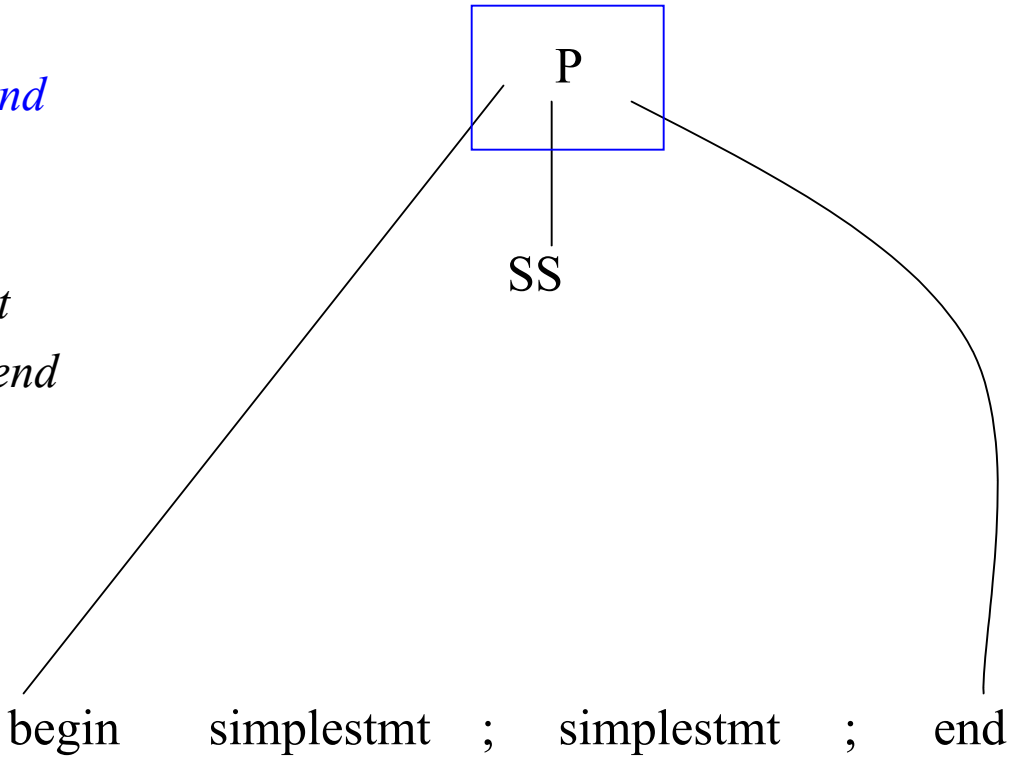
P→ *begin* SS *end*

SS → S *;* SS

SS → ε

S → *simplestmt*

S → *begin* SS *end*



Top Down (LL) Parsing

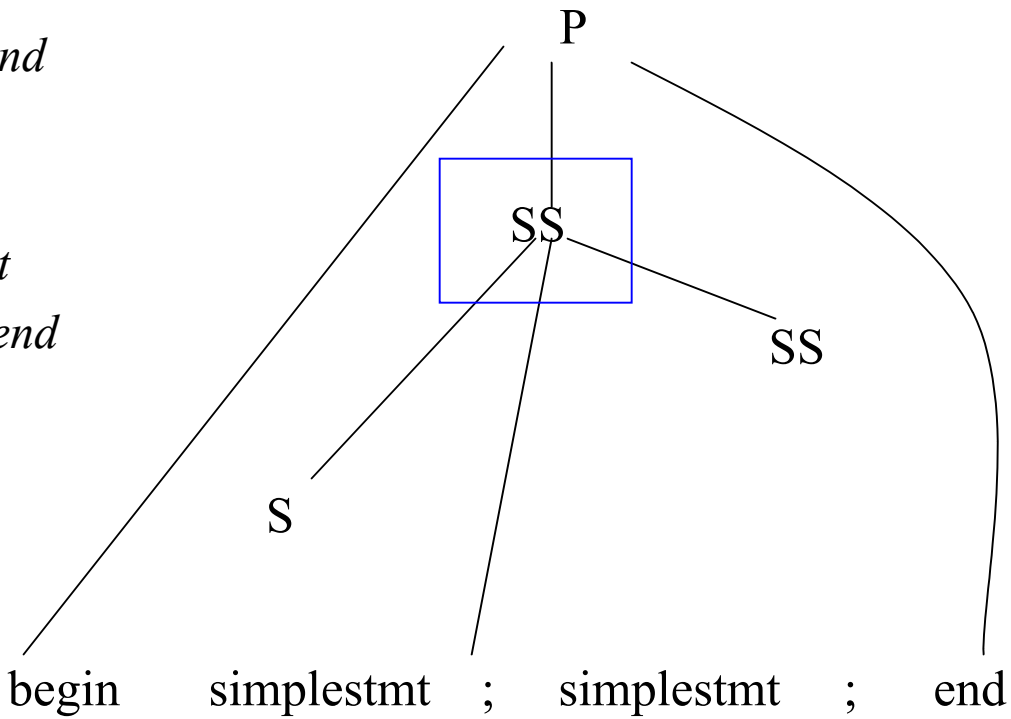
P→ *begin* SS *end*

SS → S ; SS

SS → ε

S → *simplestmt*

S → *begin* SS *end*



Top Down (LL) Parsing

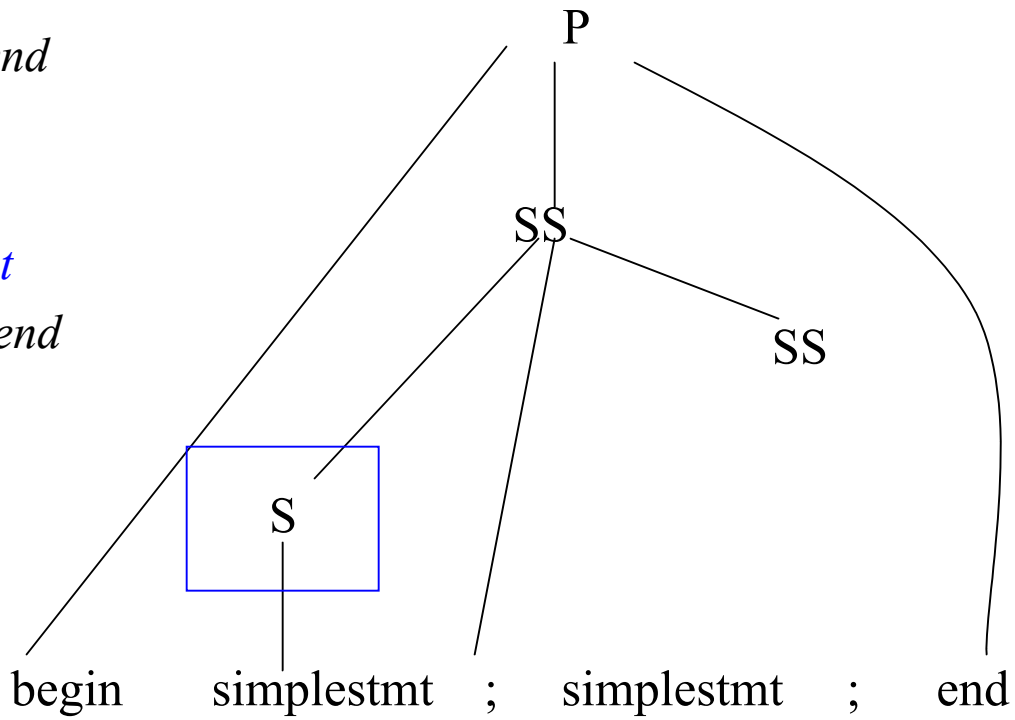
P→ *begin* SS *end*

SS → S *;* SS

SS → ε

S → *simplestmt*

S → *begin* SS *end*



Top Down (LL) Parsing

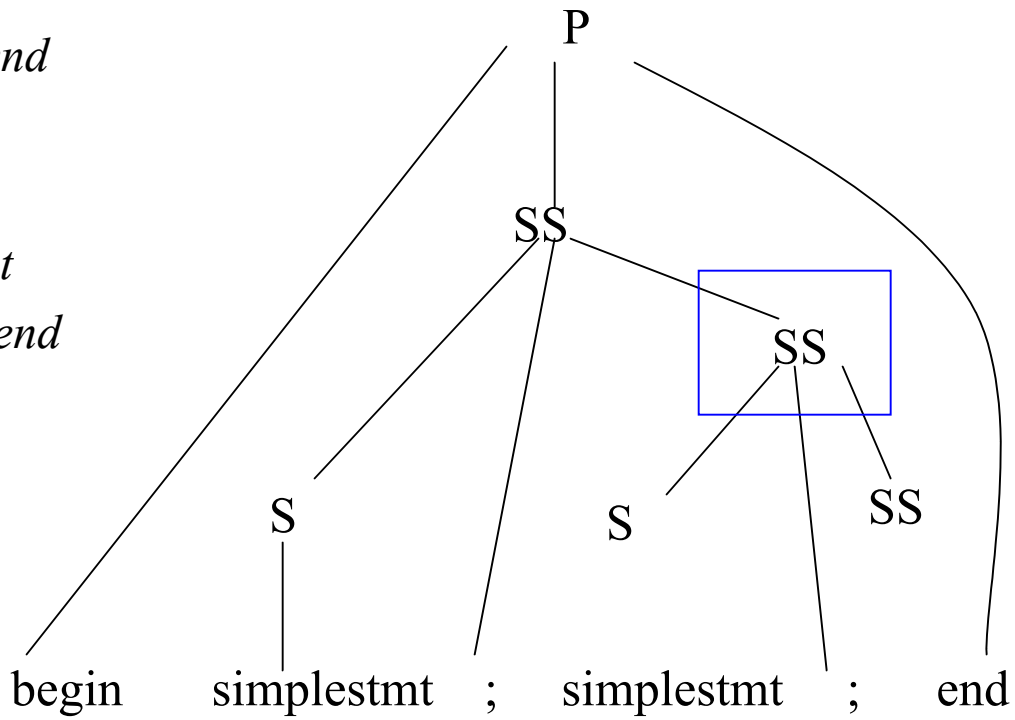
P→ *begin* SS *end*

SS → S ; SS

SS → ε

S → *simplestmt*

S → *begin* SS *end*



Top Down (LL) Parsing

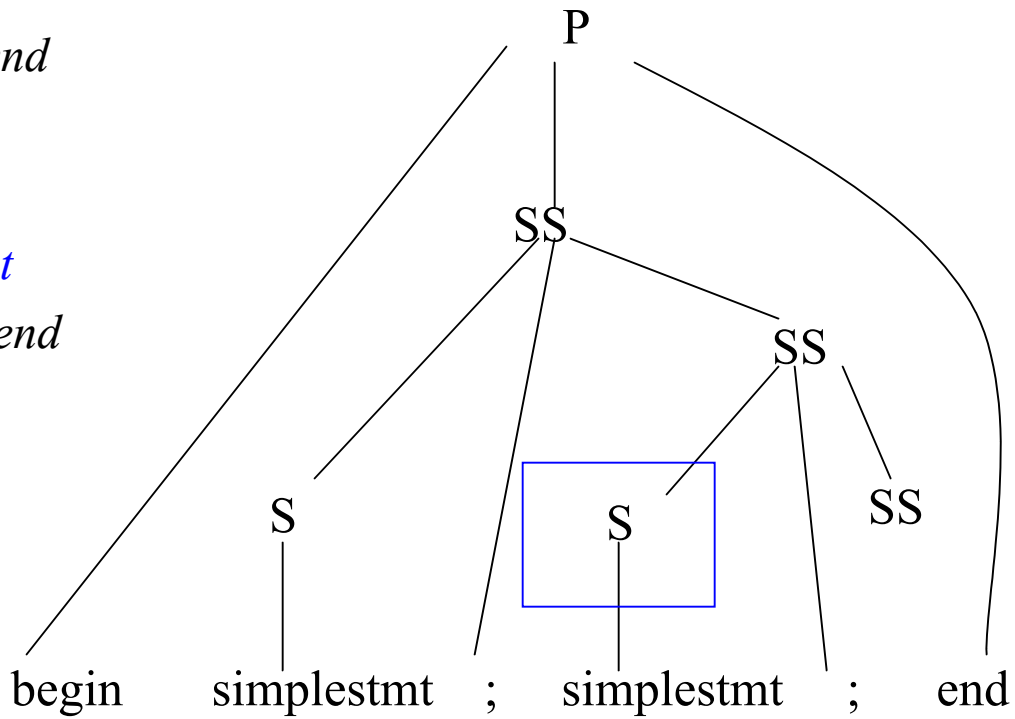
P→ *begin* SS *end*

SS → S *;* SS

SS → ε

S → *simplestmt*

S → *begin* SS *end*



Top Down (LL) Parsing

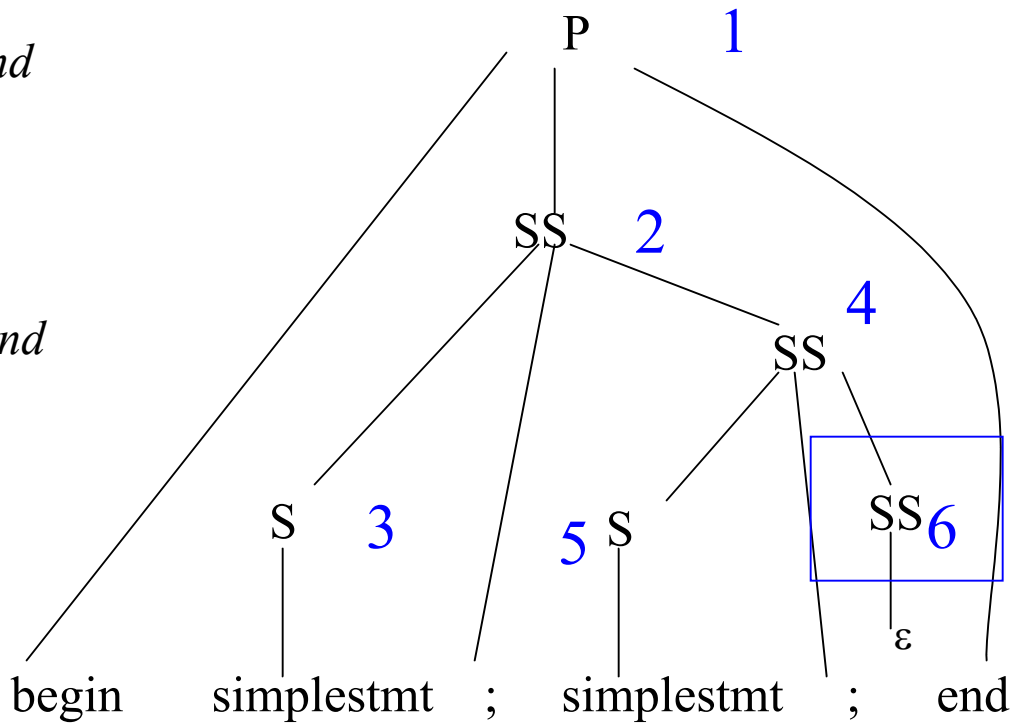
P→ *begin* SS *end*

SS → S *;* SS

SS → ε

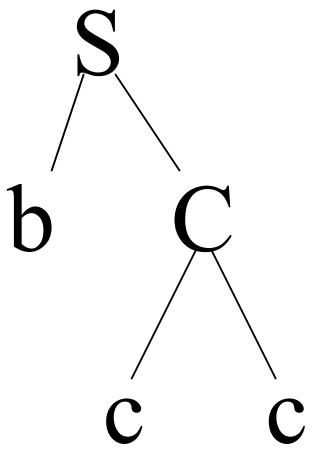
S → *simplestmt*

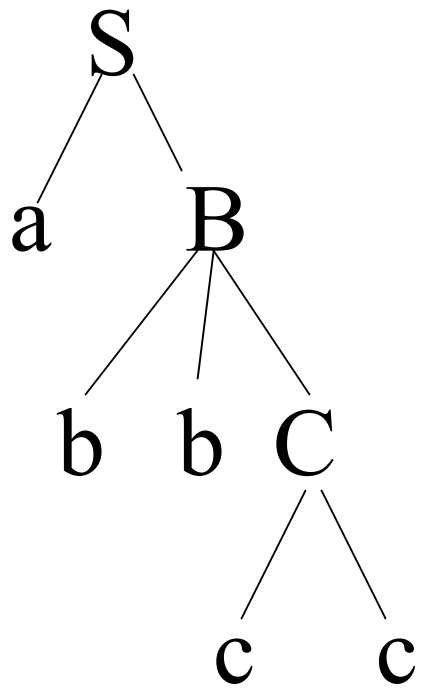
S → *begin* SS *end*



Grammar

S → a B
  | b C
B → b b C
C → 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 → …)
    - 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 → 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");
}

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

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

S → a B

S → b C

B → b b C

C → c c

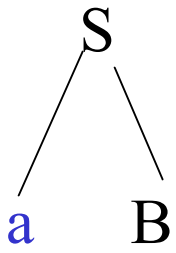
Parsing abbcc

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



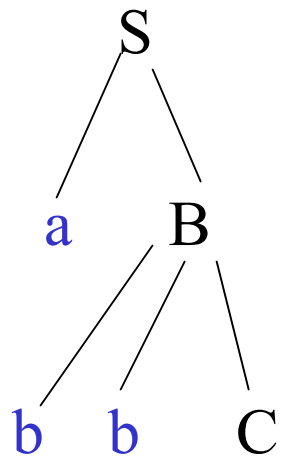
Remaining input: bbcc

Call B() from A():

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

B → b b C

Parsing abbcc



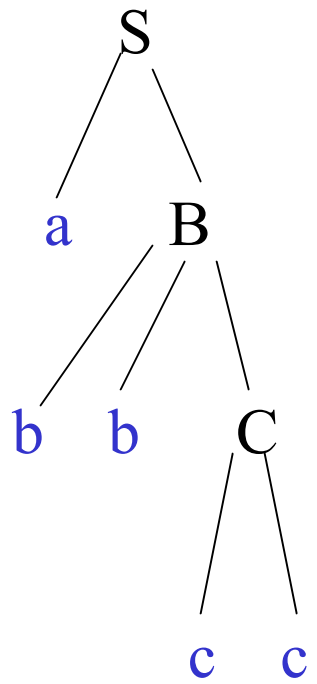
Remaining input: cc

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:
- PREDICT($A \rightarrow \alpha$)

  $= (\text{FIRST}(\alpha) - \{\varepsilon\}) \cup \text{FOLLOW}(A)$ if ε in FIRST(α)

  $= \text{FIRST}(\alpha)$ if ε not in FIRST(α)

NOTE: ε 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 → T E'	= FIRST(T) = {(,id}
E' → + T E'	{+}
E' → ε	= FOLLOW(E') = {$,)}
T → F T'	= FIRST(F) = {(,id}
T' → * F T'	{*}
T' → ε	= FOLLOW(T') = {+,$,)}
F → id	{id}
F → (E)	{(}

FIRST(F) = {(,id}
FIRST(T) = {(,id}
FIRST(E) = {(,id}
FIRST(T') = {*,ε}
FIRST(E') = {+,ε}
FOLLOW(E) = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T) = {+$,)}
FOLLOW(T') = {+,$,)}
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");
}
```

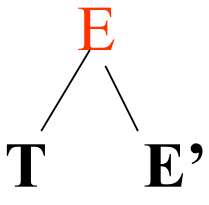
```
T_prime() {
  if (lookahead in {*}) {match(*); F(); T_prime();}   T' → * F T'
  else if (lookahead in {+,),end_of_file}) return;          T' → ε
  else error("T_prime expecting *, ) or end of file"); }

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

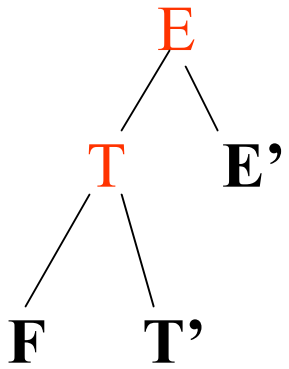
Parsing a + b * c



Remaining input: a+b*c

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

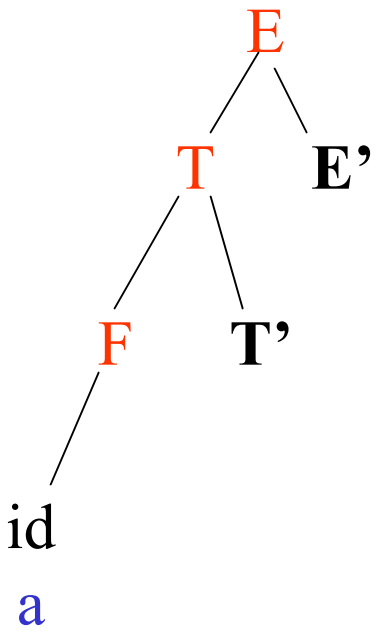
Parsing a + b * c



Remaining input: a+b*c

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

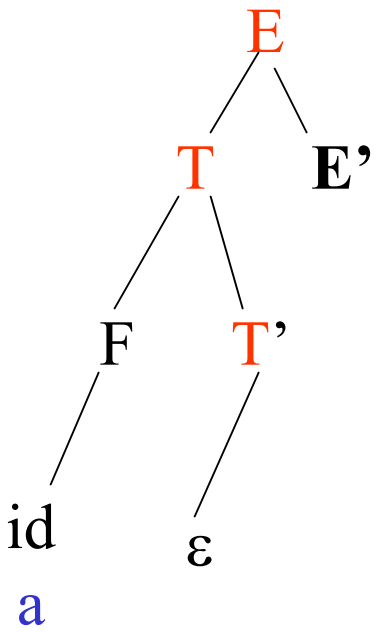
Parsing a + b * c



Remaining input: +b*c

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

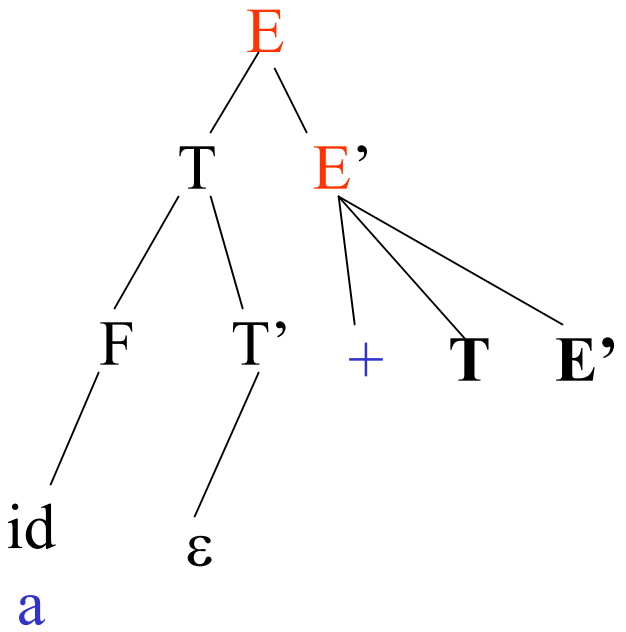
Parsing a + b * c



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"); }
}
```

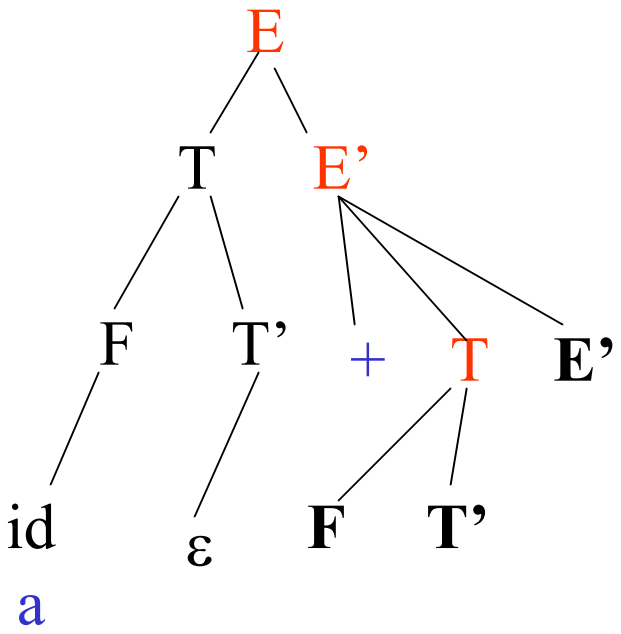
Parsing a + b * c



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"); }
}
```

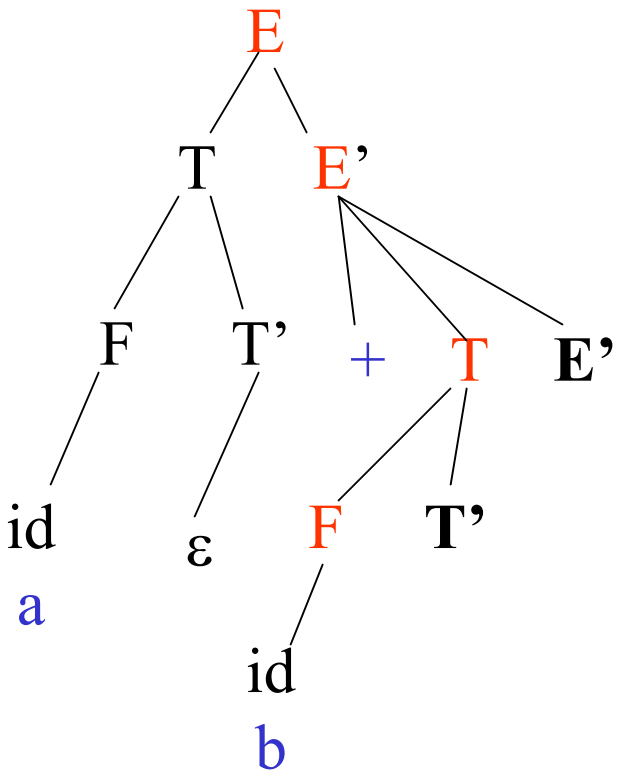
Parsing a + b * c



Remaining input: b*c

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

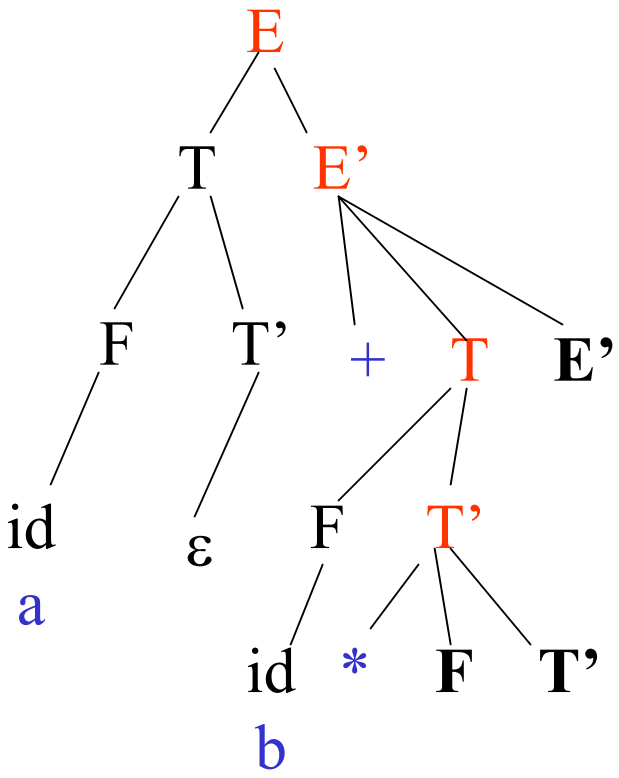
Parsing a + b * c



Remaining input: *c

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

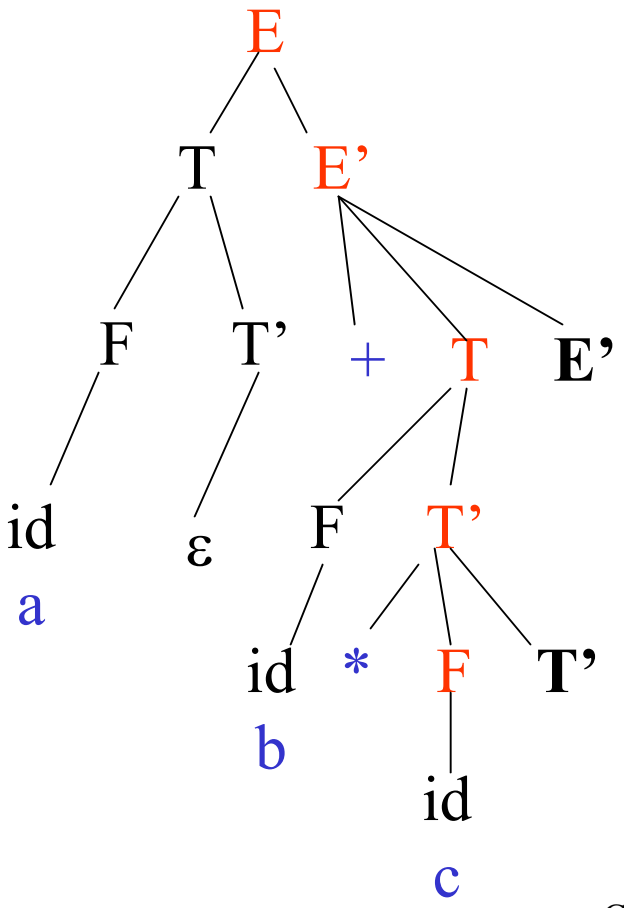
Parsing a + b * c



Remaining input: 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"); }
}
```

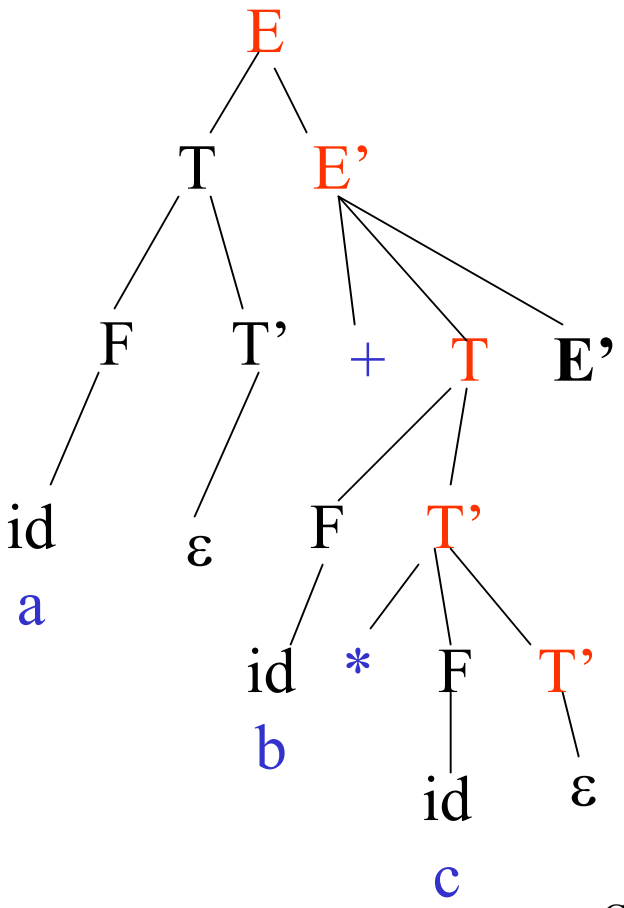
Parsing a + b * c



Remaining input:

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

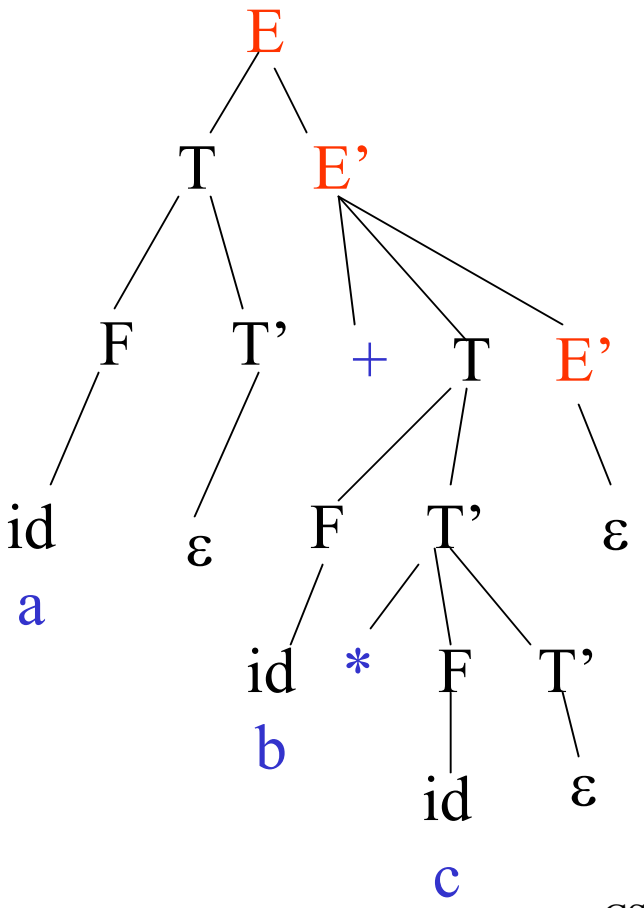
Parsing a + b * c



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"); }
}
```

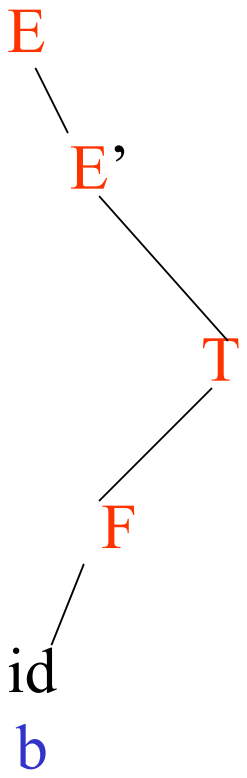
Parsing a + b * c



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 x V_t → production P or error

1. For all productions A → α do

   For each terminal t in Predict(A →α),

   T[A][t] = A → α

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][lookahead] = X → Y1 Y2 …Ym
            push(Ym) … push (Y1)
        else error()
 until X = $ token
```

similar to ‘match’

similar to ‘mimic’

Example

NT/T	+	*	(	)	ID	$
E						
E’						
T						
T’						
F						

Production	Predict
1: E → T E’	{(,id}
2: E’ → + T E’	{+}
3: E’ → ε	{$,)}
4: T → F T’	{(,id}
5: T’ → * F T’	{*}
6: T’ → ε	{+,$,)}
7: F → id	{id}
8: F → (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 → T E’	{(,id}
2: E’ → + T E’	{+}
3: E’ → ε	{$,)}
4: T → F T’	{(,id}
5: T’ → * F T’	{*}
6: T’ → ε	{+,$,)}
7: F → id	{id}
8: F → (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’

Assume E is the start symbol

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’

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 → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → 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 → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε

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
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + 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 → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + 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 → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match
$E’ T	b*c$	T → 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 → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match
$E’ T	b*c$	T → F T’
$E’T’F	b*c$	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 → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match
$E’ T	b*c$	T → F T’
$E’T’F	b*c$	F → id
$E’T id	b*c$	match

Parsing a + b * c

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T		T → F T’
$E’T’F		F → id
$E’T’id		match
$E’T’	+b*c$	T’ → ε
$E’		E’ → + T E’
$E’T+		match
$E’T	b*c$	T → F T’

Stack	Input	Action
$E’T’F		F → id
$E’T’id		match
$E’T’	*c$	T’ → * F T’
$E’T’F*		match
$E’T’F	c$	F → id
$E’T’id		match
$E’T’	$	T’ → ε
$E’		E’ → ε
$		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 → E + T	= FIRST(E) = {(,id}
E → T	= FIRST(T) = {(,id}
T → T * F	= FIRST(T) = {(,id}
T → F	= FIRST(F) = {(,id}
F → id	= {id}
F → (E)	= {(}

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

Two problems: E and T

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

- Eliminate common prefixes

  Ex: A → B a C D | B a C E
- Transform left recursion to right recursion

  Ex: E → E + T | T

Eliminate Common Prefixes

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

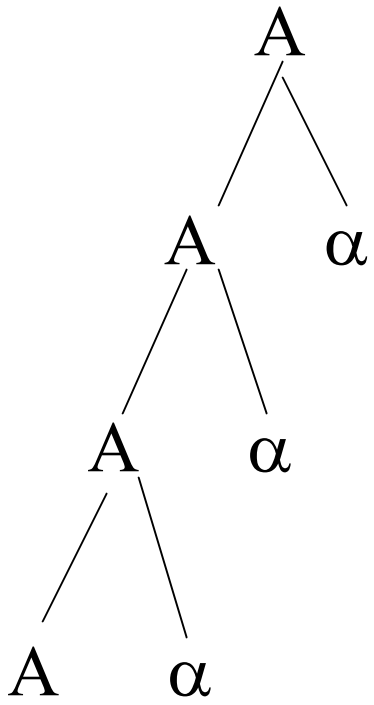
Can become:

$A \rightarrow \alpha\ A'$

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

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

Why is left recursion a problem?



Remove Left Recursion

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

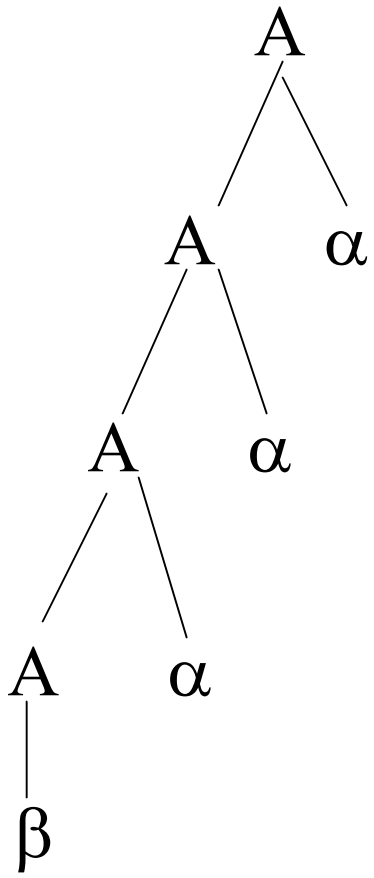
becomes

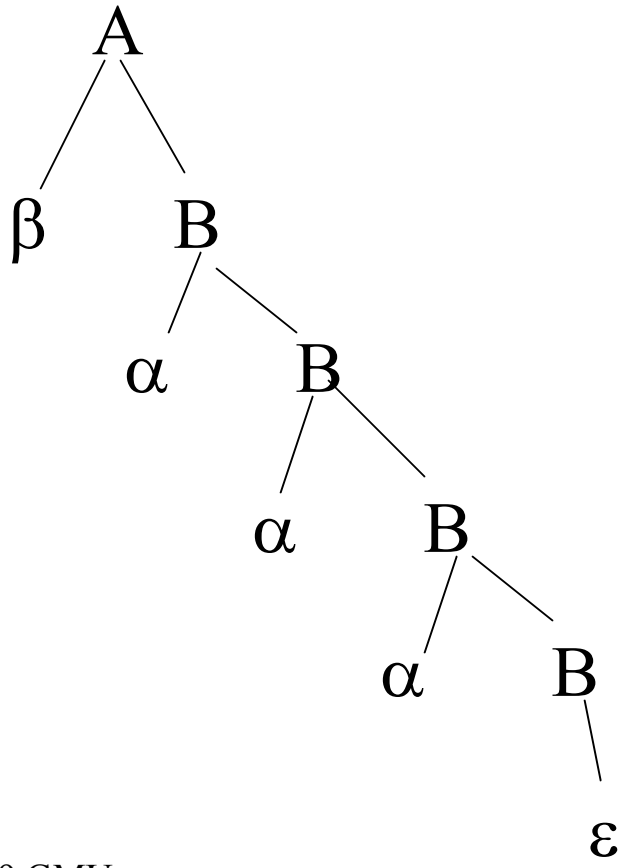
$A \rightarrow \beta_1\ A' \mid \beta_2\ A' \mid \ldots$

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

The left recursion becomes right recursion

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





Expression Grammar

- $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

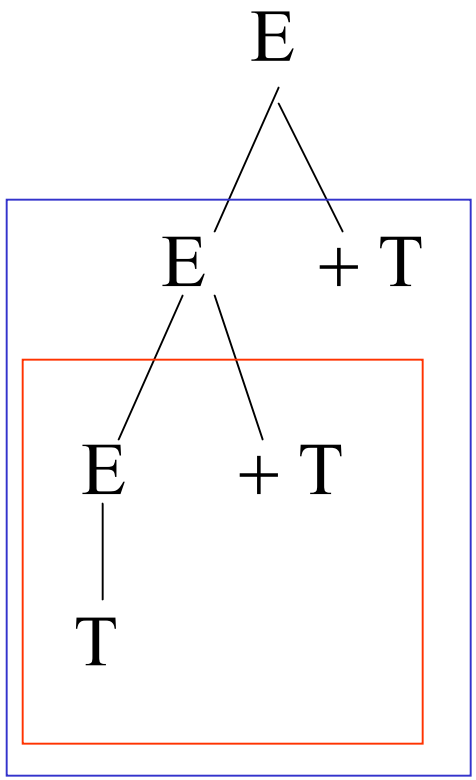
  $F \rightarrow id \mid (E)$ NOT LL(1)
- Eliminate left recursion:

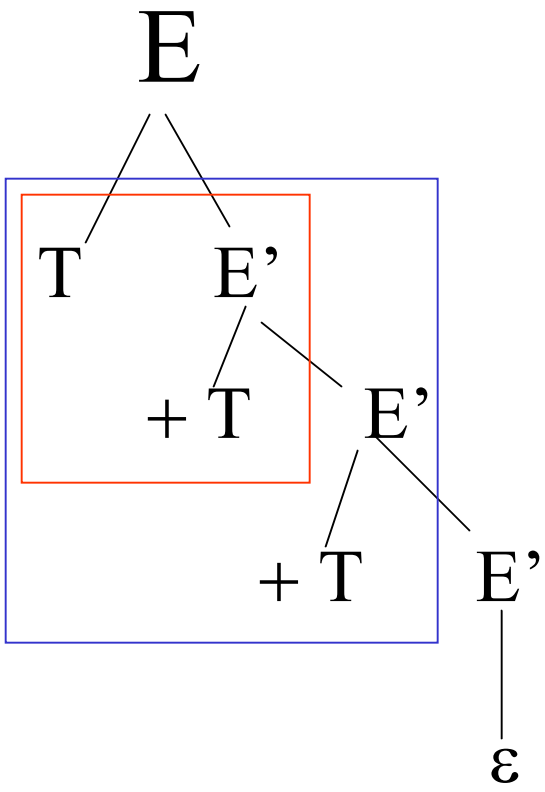
  $E \rightarrow T\ E',\quad E' \rightarrow + T\ E' \mid \varepsilon$

  $T \rightarrow F\ T',\quad T' \rightarrow * F\ T' \mid \varepsilon$

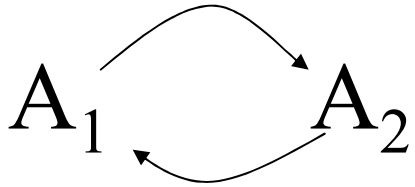
  $F \rightarrow id \mid (E)$

E → E + T | T becomes E → T E', E' → + T E' | ε





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:

$A_1 \rightarrow A_2\ a \mid b$

$A_2 \rightarrow b\ c\ A_3$

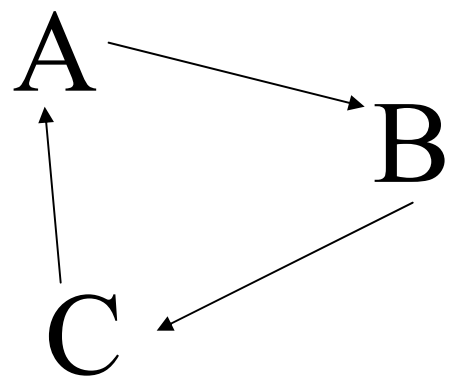
$A_3 \rightarrow a\ c\ A_3 \mid d\ A_3 \mid \varepsilon$

Example

- A → B c | d

  B → C f | B f

  C → A e | g



- Rewrite: replace C in B

  B → A e f | g f | B f

- Rewrite: replace A in B

  B → B c e f | d e f | g f | B f

- Now grammar is:

  A → B c | d

  B → B c e f | d e f | g f | B f

  C → A e | g

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

  A → B c | d

  B → d e f B' | g f B'

  B' → c e f B' | f B' | ε

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”