Compilation

0368-3133

Tutorial 2:

Top-Down Parsing

Syntax Analysis

- In lexical analysis we break the input into “words”
- In syntax analysis we check that the words form “legal sentences”

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f(int a) {
      if ((8)) {

      }
}
```

Yes

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f(int a) {
    if ((8))) {

    }
}
```

Fail

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
if ((8)) {

}
```

Fail

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f() {
        int a[];
}
```

Fail

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f() {
        int a[10.0];
}
```

Fail

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f(int a[]) {

}
```

Yes

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f() {
        int i = 0;
        int j = 1;
        j + i;
}
```

Yes

Warm-Up

- Does the input code successfully pass the ***syntax analysis*** phase?

```
void f() {
        int i = 0;
        j + i;
        int j = 1;
}
```

Yes

Context Free Grammar

- A set of terminals T and a set of non-terminals V
- Production rules of the form
  - $A \rightarrow a_1 a_1 \dots a_n$
  - $A \ni V$, $a_i \ni T \cup V$
- Starting symbol S:
  - $S \rightarrow a_1 a_1 \dots a_n$

Context Free Grammar – Questions

- Are there languages which **have no CFG**?

  Yes!

- Can we have **multiple** CFG's describing the same language?

  Yes!

Context Free Grammar – Example

$S \rightarrow c$

$S \rightarrow aSb$

- Which words belong to this grammar?
  - c, acb, aacbb, aaacbbb, …

Balanced parentheses

Does the language of balanced parentheses have a CFG?

Balanced parentheses

Does the language of balanced parentheses have a CFG? **Yes!**

$$S \rightarrow INT$$

$$S \rightarrow (S)$$

Top-Down Parsers

- Some languages have a predictive parser
- Scans the input from left to right
- Determines next production rule to apply according to the current token
- Can be implement using recursive descent

Recursive Descent Parser – Example

The language of balanced parentheses:

$$S \rightarrow INT$$

$$S \rightarrow (S)$$

has a predictive parser

Recursive Descent Parser – Example

S → INT
S → (S)

```
void parse_S() {
  switch (token) {
  case INT:
    match(INT);
    break;
  case L_PAREN:
    match(L_PAREN);
    parse_S();
    match(R_PAREN);
    break
  default:
    // error
  }
}
```

```
void parse() {
  parse_S();
  if (token != EOF)
    // error
}
```

```
void match(int expected) {
  if (token == expected) {
      token = next_token();
  } else {
    // error
  }
}
```

Recursive Descent Parser – Example

Call trace input (7)

```
parse_S
	match // match with '('
	parse_S
		match // match with '7'
	match // match with ')'
```

Recursive Descent Parser – Example

Call trace input ((7)

```
parse_S
    match // match with '('
    parse_S
        match // match with '('
        parse_S
            match // match with '7'
        match // match with ')'
    match // error, expecting ')'
```

Language of Balanced Parentheses #2

- Find a CFG for a language with the 3 kinds of parentheses:
  - (), [], {}
- Contains string of the form:
  - (([][]{}))[]
  - [()]
- Not allowing:
  - (()){, (({}))}

Language of Balanced Parentheses #2

$S \rightarrow (S) S$

$S \rightarrow [S] S$

$S \rightarrow \{ S \} S$

$S \rightarrow \varepsilon$

Language of Balanced Parentheses #2

```
void parse_S() {
  switch (token) {
  case L_PAREN:
    parse_S1();
    break;
  case L_BRACKET:
    parse_S2();
    break;
  case L_BRACE:
    parse_S3();
    break;
  default:
    break;
  }
}
```

```
void parse_S1() {
  match(L_PAREN);
  parse_S();
  match(R_PAREN);
  parse_S();
}
void parse_S2() {
  match(L_BRACKET);
  parse_S();
  match(R_BRACKET);
  parse_S();
}
void parse_S3() {
  match(L_BRACE);
  parse_S();
  match(R_BRACE);
  parse_S();
}
```

$S \rightarrow (S) S$

$S \rightarrow [S] S$

$S \rightarrow \{ S \} S$

$S \rightarrow \varepsilon$

LL(1) Parser

- Scans the input from left to right
- Produces the leftmost derivation
- Uses a lookahead of 1 token

Calculator Language

A language with binary operators (+,-,*,/) and numbers:

- 1
- 1+1
- (1+1)*(7/2)
- 2+1-7

Calculator Language

S → INT

S → S + S

S → S – S

S → S * S

S → S / S

S → (S)

- Does this grammar have a LL(1) parser?

Calculator Language

- The previous CFG cannot be parsed by an LL(1) parser
- If we see 1, what rule should we choose?
  - The input can continue with 1 + …, 1 - …, 1 * …, 1 / …
  - No way to predict the next derivation rule

Left Recursion

- Why it happens?
- Consider the rules
  - S → INT
  - S → S + S

- Need to eliminate left recursion

Left Recursion Elimination

$X \rightarrow a$

$X \rightarrow X\ b$

- The language contains: a, ab, abb, abbb,…

- Define an alternative CFG:

$X \rightarrow a\ Y$

$Y \rightarrow b\ Y \mid \varepsilon$

Left Recursion Elimination

- In general, if we have:

  $X \rightarrow a_1 \mid a_2 \mid \ldots$

  $X \rightarrow X\ b_1 \mid X\ b_2 \mid \ldots$

- We will rewrite as follows:

  $X \rightarrow a_1Y \mid a_2Y \mid \ldots$

  $Y \rightarrow b_1Y \mid b_2Y \mid \ldots \mid \varepsilon$

Calculator Language

- Before left recursion elimination:

  $S \rightarrow INT \mid (S)$

  $S \rightarrow S + S \mid S - S \mid S * S \mid S / S$

- The resulting CFG:

  $S \rightarrow INT\ T \mid S \rightarrow (S)\ T$

  $T \rightarrow +\ S\ T \mid -\ S\ T \mid *\ S\ T \mid /\ S\ T \mid \varepsilon$

In general, if we have:

$X \rightarrow a_1 \mid a_2 \mid \ldots$

$X \rightarrow X\ b_1 \mid X\ b_2 \mid \ldots$

We will rewrite as follows:

$X \rightarrow a_1 Y \mid a_2 Y \mid \ldots$

$Y \rightarrow b_1 Y \mid b_2 Y \mid \ldots \mid \varepsilon$

Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

T → + S T

T → – S T

T → * S T

T → / S T

T → ε

Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

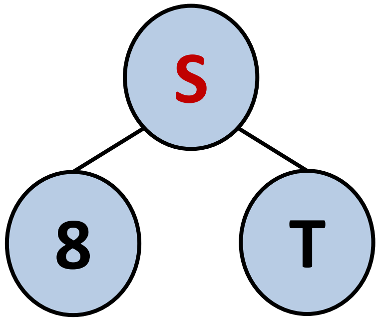
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

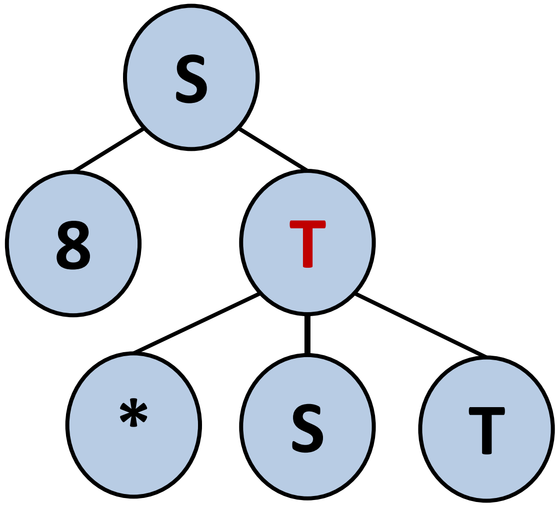
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

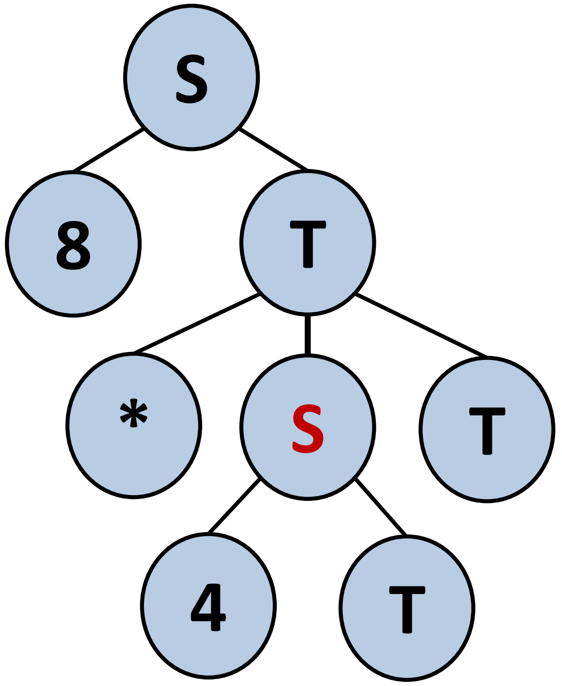
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

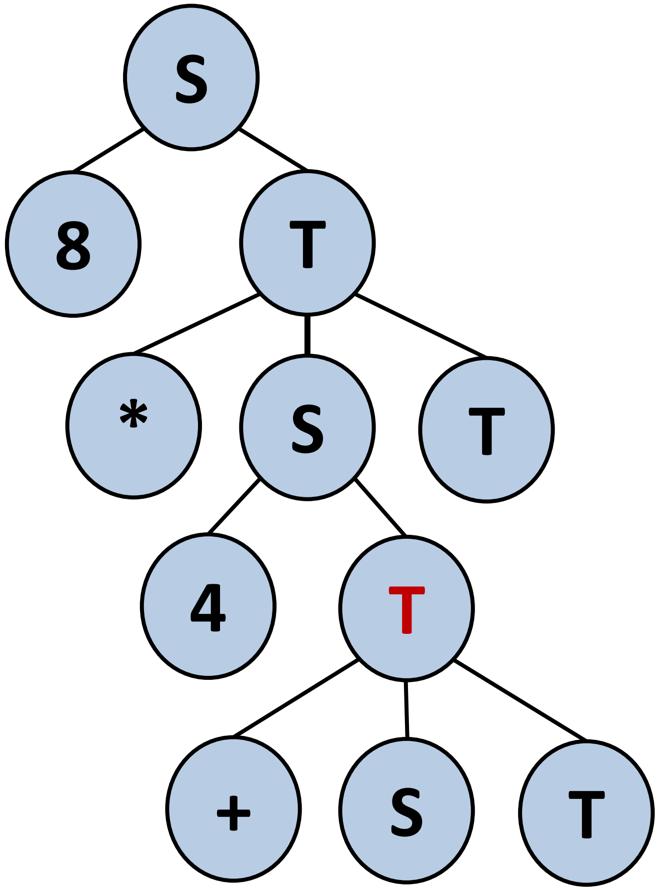
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

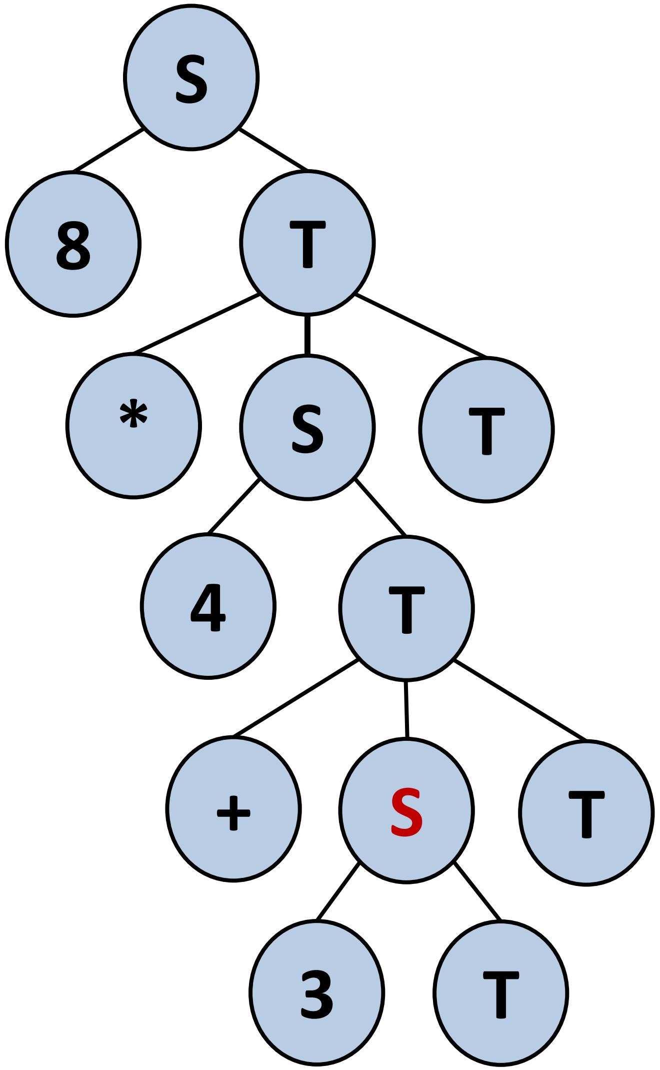
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

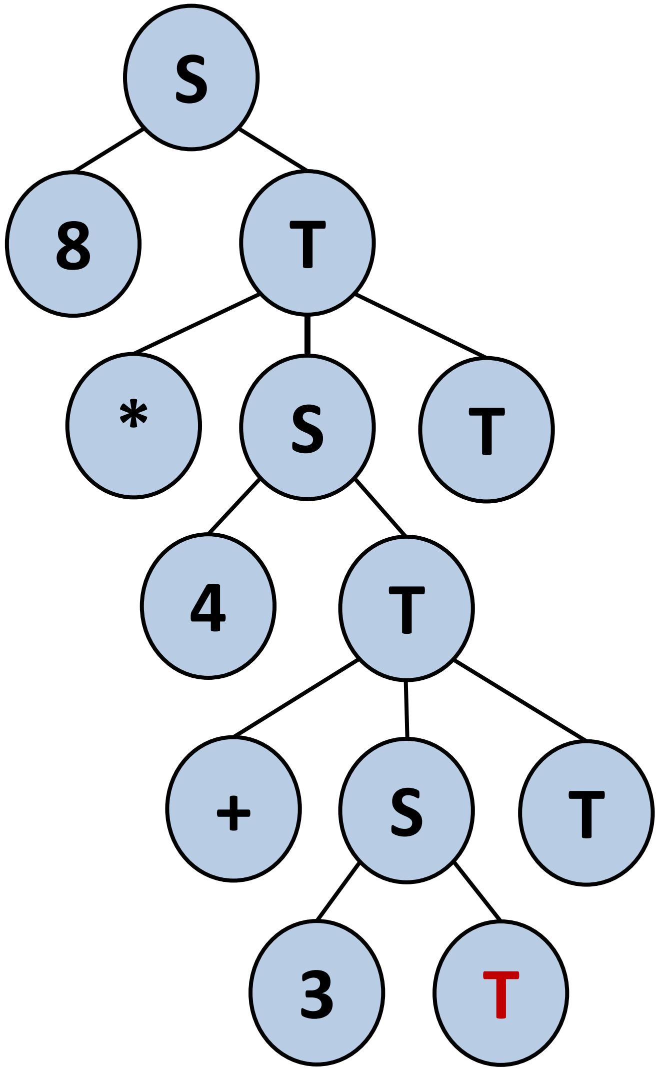
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

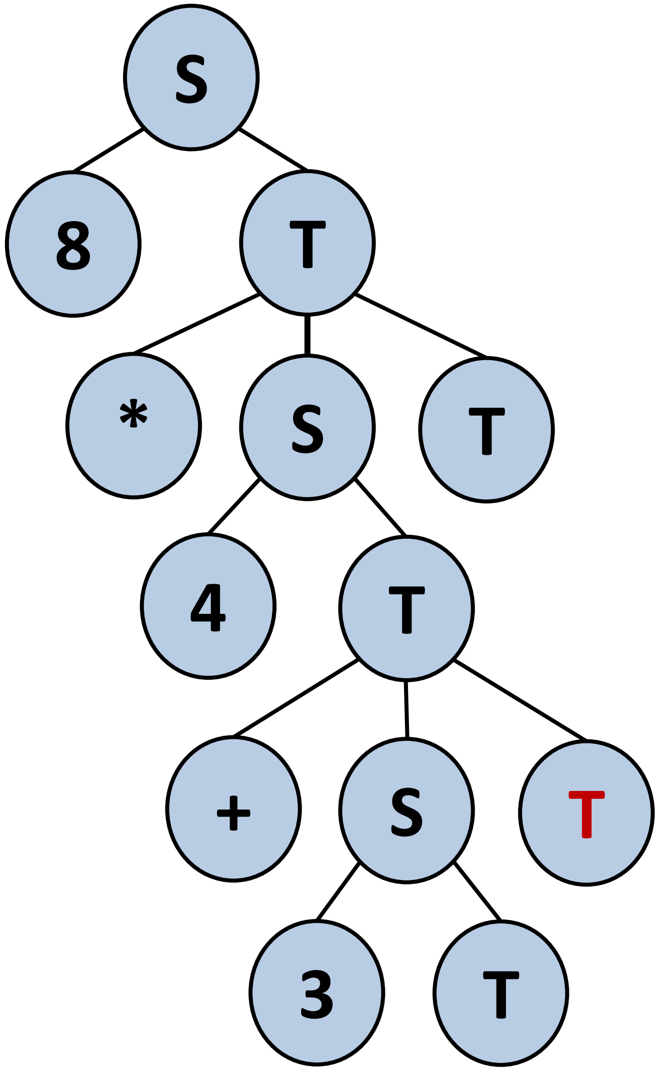
T → + S T

T → – S T

T → * S T

T → / S T

T → ε



Syntax Tree

- Which rules are applied for the expression 8 * 4 + 3?

S → INT T

S → (S) T

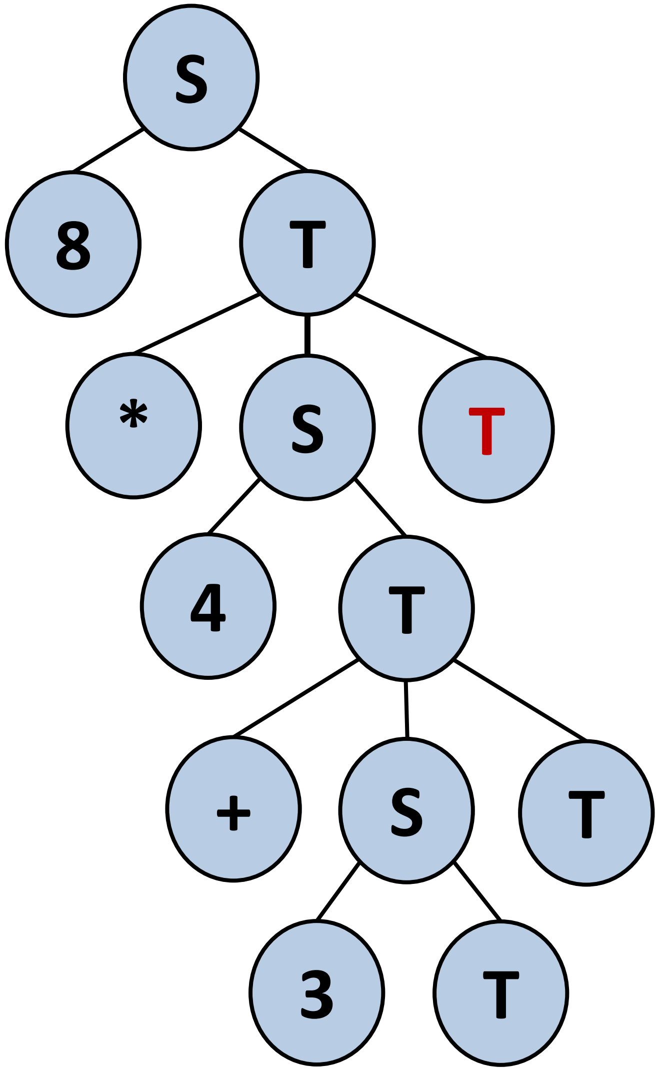
T → + S T

T → – S T

T → * S T

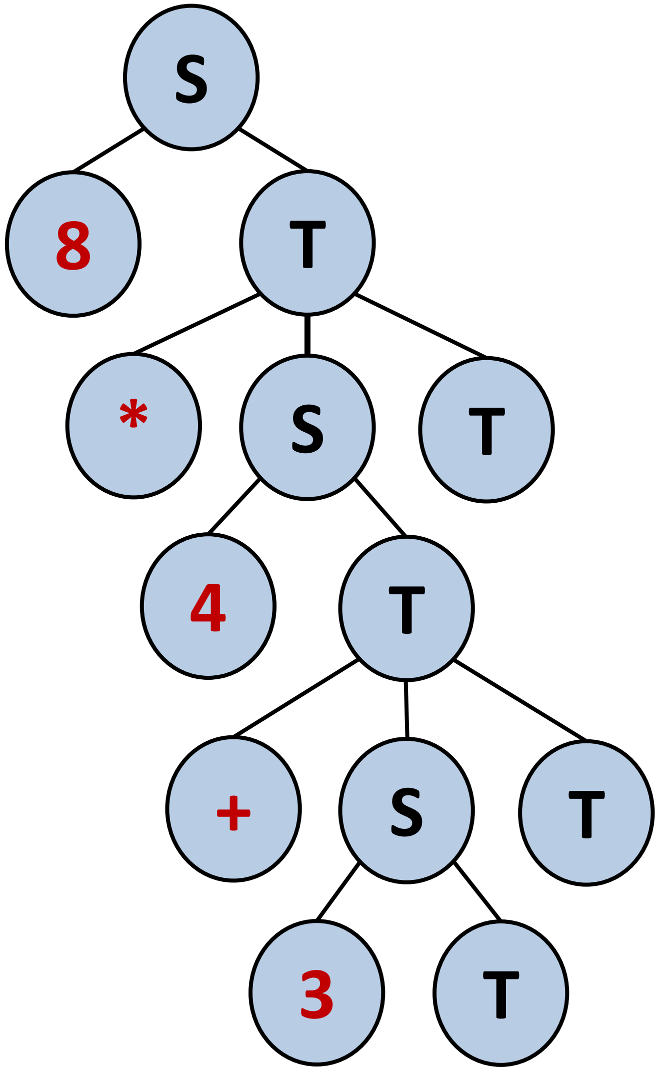
T → / S T

T → ε



Operator Precedence

- Our CFG does not contain information about operator precedence!
- The expression 8 * 4 + 3 may be interpreted as 8 * (4 + 3)
- We need to find another grammar…



Operator Precedence

$S \rightarrow S + T$

$S \rightarrow S - T$

$S \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow INT$

$F \rightarrow (S)$

But we have left recursion so no LL(1) parser

Eliminating Left Recursion

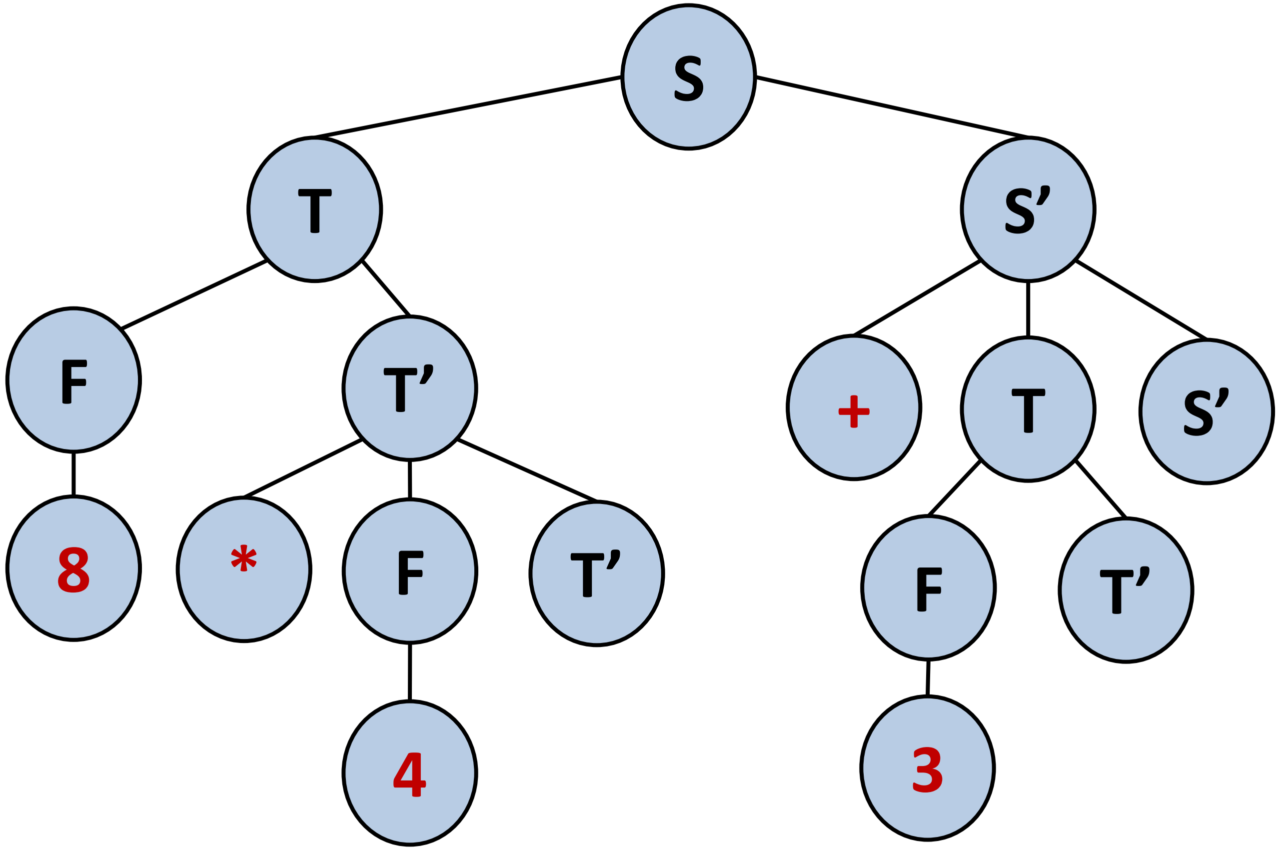
S → T S'
S' → + T S'
S' → – T S'
S' → ε

T → F T'
T' → * F T'
T' → / F T'
T' → ε

F → INT
F → (S)

Operator Precedence

- With the new CFG, the derivation tree for 8 * 4 + 3:



Left Factoring

- Left recursion was an issue, are there other issues?

- What about the next grammar:

$$\left.\begin{array}{l} E \rightarrow \text{if (E) then } E \\ E \rightarrow \text{if (E) then } E \text{ else } E \end{array}\right\} \text{both start with if}$$

$E \rightarrow INT$

Left Factoring

E → if (E) then E

E → if (E) then E else E

E → INT

⇒

E → if (E) then E X

X → ε

X → else E

E → INT

Nullable Rules

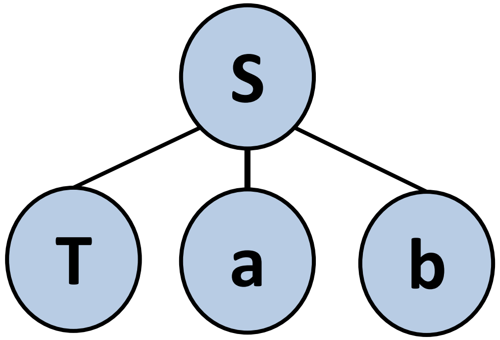
- Consider the following grammar:

S → Tab

T → a | ε

- No left recursion, no left factoring…
- But does it have a LL(1) parser?

Nullable Rules



- Consider the following grammar:

S → Tab

T → a | ε

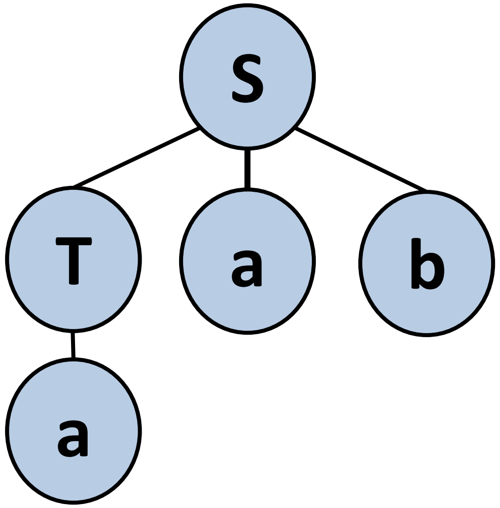
- If the first symbol is a, we can't predict what rule to choose:
  - If we choose T → a, then it will fail to parse the input ab

Nullable Rules

- Consider the following grammar:

S → Tab

T → a | ε



- If the first symbol is a, we can't predict what rule to choose:
  - If we choose T → ε, then it will fail to parse the input aab

Nullable Rules

- We can substitute T with its possible derivations

$S \rightarrow Tab$
$T \rightarrow a \mid \varepsilon$

$\Rightarrow$

$S \rightarrow ab$
$S \rightarrow aab$

- Are we done?

Nullable Rules

- We can substitute T with its possible derivations

S → Tab T → a | ε	⇨	S → ab S → aab	⇨	S → a X X → b | ab

- Are we done?
- We still need to perform left factoring

Back to Recursive Descent

```
void A() {
    Choose an A-production A → X1X2…Xk
    for (i = 1…k) {
        if (Xi is a nonterminal) call function Xi()
        else call match(Xi)
    }
}
```

How to choose the next production rule??

How to Choose the Next Production Rule?

- We define a select function to help choose the next production rule
- Maps each production rule to a set of terminals
- To pick the next rule look at the next input token
- Choose the production whose set contains the token

- We do this by computing two types of sets:
  - FIRST sets
  - FOLLOW sets

FIRST Set

For a sequence of symbols $\alpha \in (V \cup T)^*$:

$$\text{FIRST}(\alpha) = \text{all terminals (and } \varepsilon\text{) that } \alpha \text{ can start with}$$

Formal definition:

$$\text{FIRST}(\alpha) = \{\, t \in T \mid \alpha \rightarrow^* t\beta \,\} \cup \{\varepsilon \mid \alpha \rightarrow^* \varepsilon \,\}$$

- Note that for $t \in T$, $\text{FIRST}(t) = \{t\}$

Computing FIRST Sets

Initialization: for all $A \in V$ set $\text{FIRST}(A) = \{\, t \mid A \rightarrow t\beta \in P\} \cup \{\, \varepsilon \mid A \rightarrow \varepsilon \in P\}$

Repeat until no FIRST(A) changes:

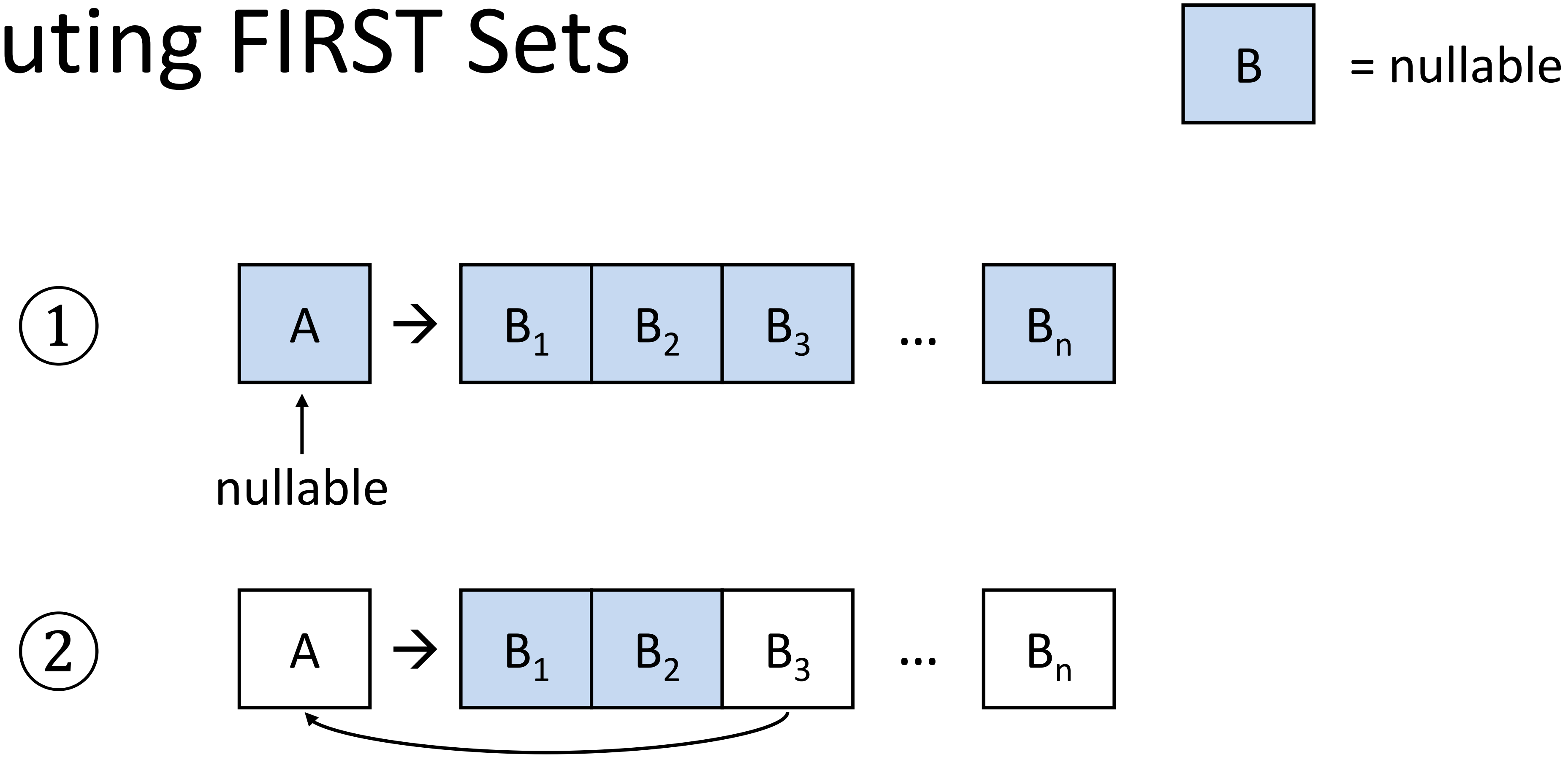
- For each $A \in V$
  - For each $A \rightarrow B_1 \ldots B_k B_{k+1} \ldots B_n \in P$:
    - If $\varepsilon \in \text{FIRST}(B_i)$ for every $1 \geq i \geq n$:
      - $\text{FIRST}(A) = \text{FIRST}(A) \cup \{\varepsilon\}$
    - For each $k = 1 \ldots n\text{-}1$ such that $\varepsilon \in \text{FIRST}(B_i)$ for every $1 \geq i \geq k$:
      - $\text{FIRST}(A) = \text{FIRST}(A) \cup (\text{FIRST}(B_{k+1}) \setminus \{\varepsilon\})$
    - $\text{FIRST}(A) = \text{FIRST}(A) \cup (\text{FIRST}(B_1) \setminus \{\varepsilon\})$

Computing FIRST Sets

B = nullable

① A → B_1 B_2 B_3 … B_n

Computing FIRST Sets



$$\text{FIRST}(A) = \text{FIRST}(A) \cup (\text{FIRST}(B_3) \setminus \{\varepsilon\})$$

Computing FIRST Sets

To compute FIRST(α) for $\alpha = X_1X_2 \ldots X_n \in (V \cup T)^*$:

- Initialize: FIRST(α) = FIRST(X_1) \ {ε}
- If $\varepsilon \in$ FIRST(X_i) for every $1 \geq i \geq n$:
  - FIRST(α) = FIRST(α) ∪ {ε}
- For each k = 1…n-1 :
  - If $\varepsilon \in$ FIRST(X_i) for every $1 \geq i \geq k$:
    - FIRST(α) = FIRST(α) ∪ (FIRST(X_{k+1})\{ε})

Computing FIRST Sets – Example

S → aB | BC | CBd

B → b | C

C → c | ε

	S	B	C
0	{a}	{b}	{c,ε}

Computing FIRST Sets – Example

$S \rightarrow aB \mid BC \mid CBd$

$B \rightarrow b \mid C$

$C \rightarrow c \mid \varepsilon$

	S	B	C
0	{a}	{b}	{c,ε}
1			

Computing FIRST Sets – Example

S → aB | **B**C | **C**Bd

B → b | C

C → c | ε

	S	B	C
0	{a}	{b}	{c,ε}
1	{a,b,c}		

FIRST(S) = FIRST(S) ∪ (FIRST(B)\{ε})

FIRST(S) = FIRST(S) ∪ (FIRST(C)\{ε})

Computing FIRST Sets – Example

S → aB | BC | **C**Bd

B → b | C

C → c | ε

	S	B	C
0	{a}	{b}	{c,ε}
1	{a,b,c}		

$$\varepsilon \in \text{FIRST}(C) \Longrightarrow \text{FIRST}(S) = \text{FIRST}(S) \cup (\text{FIRST}(B) \setminus \{\varepsilon\})$$

Computing FIRST Sets – Example

S → aB | BC | CBd

B → b | **C**

C → c | ε

	S	**B**	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c}	

FIRST(B) = FIRST(B) ∪ (FIRST(C)\{ε})

Computing FIRST Sets – Example

$S \rightarrow aB \mid BC \mid CBd$

$B \rightarrow b \mid \mathbf{C}$

$C \rightarrow c \mid \varepsilon$

	S	**B**	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c,ε}	

$$\varepsilon \in \text{FIRST}(C) \Longrightarrow \text{FIRST}(B) = \text{FIRST}(B) \cup \{\varepsilon\}$$

Computing FIRST Sets – Example

S → aB | BC | CBd

B → b | **C**

C → c | ε

	S	**B**	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c,ε}	{c,ε}

Computing FIRST Sets – Example

S → aB | **BC** | CBd

B → b | C

C → c | ε

	S	B	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c,ε}	{c,ε}
2	{a,b,c,ε}		

$$\varepsilon \in \text{FIRST}(B) \land \varepsilon \in \text{FIRST}(C) \Longrightarrow \text{FIRST}(S) = \text{FIRST}(S) \cup \{\varepsilon\}$$

Computing FIRST Sets – Example

$S \rightarrow aB \mid BC \mid \mathbf{CB}d$

$B \rightarrow b \mid C$

$C \rightarrow c \mid \varepsilon$

	S	B	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c,ε}	{c,ε}
2	{a,b,c,d,ε}		

$$\varepsilon \in FIRST(C) \wedge \varepsilon \in FIRST(B) \Longrightarrow FIRST(S) = FIRST(S) \cup FIRST(d)$$

Computing FIRST Sets – Example

S → aB | BC | CBd

B → b | C

C → c | ε

	S	B	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c,ε}	{c,ε}
2	{a,b,c,d,ε}	{b,c,ε}	{c,ε}

Computing FIRST Sets – Example

$S \rightarrow aB \mid BC \mid CBd$

$B \rightarrow b \mid C$

$C \rightarrow c \mid \varepsilon$

	S	B	C
0	{a}	{b}	{c,ε}
1	{a,b,c}	{b,c,ε}	{c,ε}
2	{a,b,c,d,ε}	{b,c,ε}	{c,ε}
3	{a,b,c,d,ε}	{b,c,ε}	{c,ε}

FOLLOW Set

end-of-input marker

For a nonterminal $A \in V$:

FOLLOW(A) = set of terminals (and \$) that can immediately follow A

Formal definition:

$$\text{FOLLOW}(A) = \{\, t \in T \mid S \rightarrow^* \alpha A t \beta \,\} \cup \{\, \$ \mid S \rightarrow^* \alpha A \,\}$$

Computing FOLLOW Sets

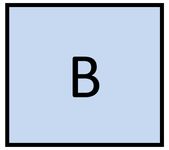
Initialization: for all $B \in V$ set $\text{FOLLOW}(B) = \{ t \in \text{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P\} \setminus \{\varepsilon\}$

Set $\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}$

Repeat until no FOLLOW(A) changes:

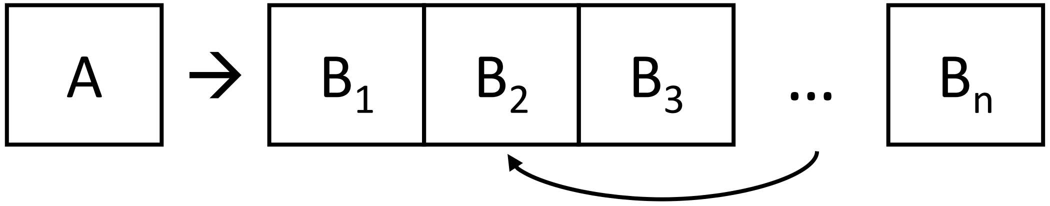
- for each $A \rightarrow \alpha B \beta \in P$
  - if $\varepsilon \in \text{FIRST}(\beta)$
    - $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

Computing FOLLOW Sets



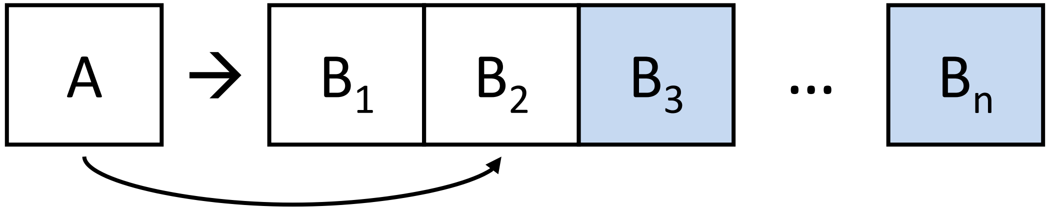
= nullable

1



$$\text{FOLLOW}(B_2) = \text{FOLLOW}(B_2) \cup (\text{FIRST}(B_3 \ldots B_n) \setminus \{\varepsilon\})$$

2



$$\text{FOLLOW}(B_2) = \text{FOLLOW}(B_2) \cup \text{FOLLOW}(A)$$

Computing FOLLOW Sets – Example

S → aB | BC | CBd

B → b | C

C → c | ε

	S	B	C
0	{$}	{d,c}	{b,c,d}

FIRST(S) = {a,b,c,d,ε}
FIRST(B) = {b,c,ε}
FIRST(C) = {c,ε}

Computing FOLLOW Sets – Example

$S \rightarrow aB \mid BC \mid CBd$

$B \rightarrow b \mid C$

$C \rightarrow c \mid \varepsilon$

	S	B	C
0	{\$}	{d,c}	{b,c,d}
1			

FIRST(S) = {a,b,c,d,ε}
FIRST(B) = {b,c,ε}
FIRST(C) = {c,ε}

for each $A \rightarrow \alpha B\beta \in P$
    if $\varepsilon \in FIRST(\beta)$
        $FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$

Computing FOLLOW Sets – Example

$S \rightarrow aB \mid \mathbf{B}C \mid CBd$

$B \rightarrow b \mid C$

$C \rightarrow c \mid \varepsilon$

	S	B	C
0	{\$}	{d,c}	{b,c,d}
1	{\$}	{d,c,\$}	

FIRST(S) = {a,b,c,d,ε}
FIRST(B) = {b,c,ε}
FIRST(C) = {c,ε}

for each $A \rightarrow \alpha B \beta \in P$
    if $\varepsilon \in \text{FIRST}(\beta)$
        $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

Computing FOLLOW Sets – Example

S → aB | BC | **C**Bd

B → b | C

C → c | ε

	S	B	C
0	{$}	{d,c}	{b,c,d}
1	{$}	{d,c,$}	{b,c,d,$}

FIRST(S) = {a,b,c,d,ε}
FIRST(B) = {b,c,ε}
FIRST(C) = {c,ε}

for each A → αBβ ∈ P
    if ε ∈ FIRST(β)
        FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(A)

Computing FOLLOW Sets – Example

$S \rightarrow aB \mid BC \mid CBd$

$B \rightarrow b \mid C$

$C \rightarrow c \mid \varepsilon$

	S	B	C
0	{\$}	{d,c}	{b,c,d}
1	{\$}	{d,c,\$}	{b,c,d,\$}
2	{\$}	{d,c,\$}	{b,c,d,\$}

FIRST(S) = {a,b,c,d,ε}
FIRST(B) = {b,c,ε}
FIRST(C) = {c,ε}

for each $A \rightarrow \alpha B \beta \in P$
  if $\varepsilon \in FIRST(\beta)$
    $FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$

Selecting the Next Rule

```
void A() {
    Choose an A-production A → X1X2…Xk
    such that token ∈ FIRST(X1X2…Xk)
        or ε ∈ FIRST(X1X2…Xk) and token ∈ FOLLOW(A)
    for (i = 1…k) {
        if (Xi is a nonterminal) call function Xi()
        else call match(Xi)
    }
}
```

LL(1) Grammar

- LL(1) parser uses the select function to choose the next rule

- A grammar has a LL(1) parser iff
  - For every two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ we have:
    - $FIRST(\alpha) \cap FIRST(\beta) = \{\}$
    - If $\varepsilon \in FIRST(\alpha)$ then $FIRST(\beta) \cap FOLLOW(A) = \{\}$
    - If $\varepsilon \in FIRST(\beta)$ then $FIRST(\alpha) \cap FOLLOW(A) = \{\}$

LL(1) Parsing is Not Always Possible ☹

- Common reasons a grammar is not LL(1) include:
  - Ambiguity
  - Left recursion
  - Needs left factoring
  - Nullable rules

LL(1) Parsing is Not Always Possible ☹

- The following grammar cannot be fixed:

$S \rightarrow A$

$S \rightarrow B$

$A \rightarrow a\ A\ b$

$A \rightarrow \varepsilon$

$B \rightarrow a\ B\ bb$

$B \rightarrow \varepsilon$

LL(1) Parsing: Is It the Best Choice?

- Grammars of real PLs are overloaded with
  - Left recursion
  - Left factoring
  - Nullable rules

- Even if we can fix it, the resulting grammar may be unreadable…