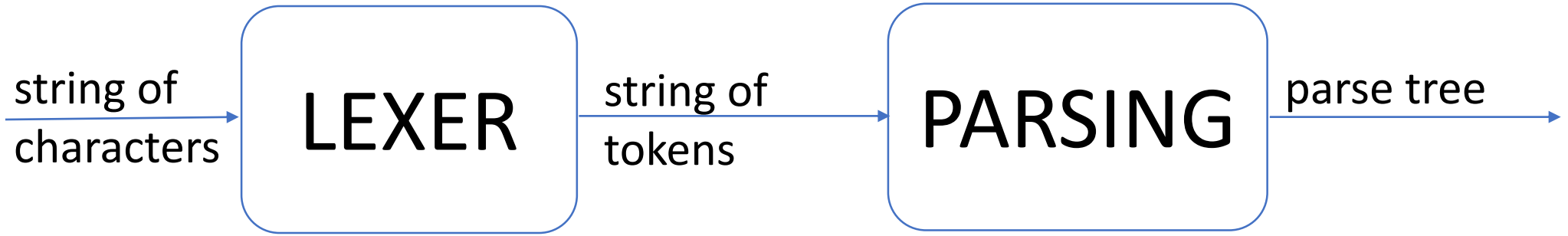
ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Chương 3
Phân tích cú pháp (Syntax Analysis)

Introduction to Parsing

- Relationship between lexical analysis and parsing.



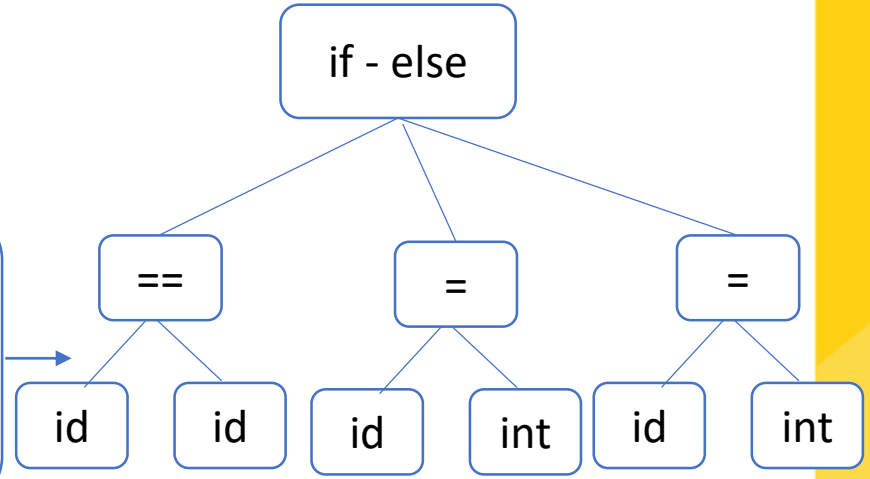
Introduction to Parsing

```
if (x==y)
   result = 1;
else
   result = 2;
```

LEXER

<key, ‘if’>
<(, ‘(‘>
<id, ‘x’>
<==, ‘==‘>
<id, ‘y’>
<), ‘)’>
<id, ‘result’>
<=, ‘=‘>
<int, ‘1’>
<pun, ‘;’>
<key, ‘else’>
<id, ‘result’>
<=, ‘=‘>
<int, ‘2’>
<pun, ‘;’>

PARSING



Context-Free Grammars (CFGs)

- A CFG consist of
  - A set of terminals Σ
  - A set of nonterminals Δ
  - A start symbol **S ($S \in \Delta$)**
  - A set of productions (rules) **R**

$$A \rightarrow B_1 B_2 \ldots . B_n,$$

$$A \in \Delta$$

$$B_i \in \Delta \cup \Sigma \cup \{\varepsilon\}$$

means A can be replaced by $B_1 B_2 \ldots . B_n$

Context-Free Grammars (CFGs)

Key idea

1. Begin with a string consisting of the start symbol "S"

2. Replace any nonterminal A in the string by a the right-hand side of some production

$$A \rightarrow B_1 \ldots B_n$$

3. Repeat (2) until there are no nonterminals in the string

Context-Free Grammars (CFGs)

Notational conventions for grammars

1. These symbols are terminals:
   (a) Lowercase letters early in the alphabet, such as a, b, c.
   (b) Operator symbols such as +, *, and so on.
   (c) Punctuation symbols such as parentheses ( ), comma ‘;’, and so on.
   (d) The digits 0, 1, ….., 9.
   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as A, B, C.
   (b) The letter S, which, when it appears, is usually the start symbol.
   (c) Lowercase, italic names such as *expr* or *stmt*.

Context-Free Grammars (CFGs)

Notational conventions for grammars

3. Uppercase letters late in the alphabet, such as *X, Y, Z,* represent *grammar symbols*; that is, either non-terminals or terminals.

4. Lowercase letters late in the alphabet, chiefly *u, v, …, z,* represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α is the body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ……., $A \rightarrow \alpha_k$
with a common head A (call them A-productions), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$
Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Context-Free Grammars (CFGs)

- Example of a Context-free Grammar

$S \rightarrow (S)$

$S \rightarrow \varepsilon$

- set of terminals $\Sigma = \{(,)\}$
- set of nonterminals $\Delta = \{S\}$
- S: S

Context-Free Grammars (CFGs)

- Example of a Context-free Grammar

expression → *expression* + *term*
expression → *expression* - *term*
expression → *term*
term → *term* * *factor*
term → *term* / *factor*
term → *factor*
factor → *(expression)*
factor → **id**

- set of terminals Σ = {+, -, *, /,(,), **id**}
- set of nonterminals Δ = {*expression*, *term*, *factor*}
- S: *expression*

Derivations

- Consider the following grammar: $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid \mathbf{id}$
- The production $E \rightarrow -E$ signies that if E denotes an expression, then -E must also denote an expression.
- The replacement of a single E by - E will be described by writing $E \Rightarrow -E$ which is read, “E derives -E"
- The production $E \rightarrow (E)$ can be applied to replace any instance of E in any string of grammar symbols by (E), e.g., $E*E \Rightarrow (E)*E$ or $E*E \Rightarrow E*(E)$
- We can take a single E and repeatedly apply productions in any order to get a sequence of replacements.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

- We call such a sequence of replacements a derivation of -(**id**) from E. This derivation provides a proof that the string -(**id**) is one particular instance of an expression.

Derivations

- Consider the following grammar: $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid \mathbf{id}$

- The string **-(id+id)** is a sentence of grammar because there is a derivation
  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$
- We write $E \Rightarrow^{+} -(\mathbf{id}+\mathbf{id})$ to indicate that **-(id+id)** can be derived from E.

- In *leftmost derivations*, the leftmost nonterminal in each sentential is always chosen.

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$

- In *rightmost derivations*, the rightmost nonterminal is always chosen.

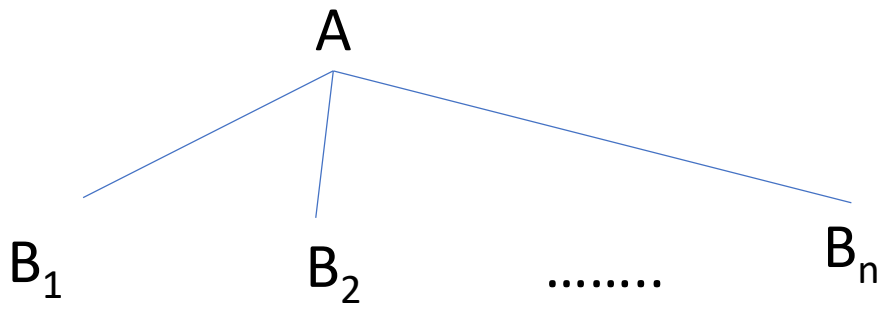
$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$

Parse Trees and Derivations

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- A parse tree has
  - Terminals at the leaves
  - Nonterminals at the interior nodes
- Each interior node of a parse tree represents the application of a production
  - The interior node is labeled with the nonterminal (A) in the head of the production
  - The children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not.

Parse Trees and Derivations

- A derivation can be drawn as a tree
  - Start symbol is the root of the tree
  - For a production $A \rightarrow B_1B_2\ldots.B_n$ add children $B_1B_2\ldots.B_n$ to node A



Parse Trees and Derivations

- Note that *leftmost derivations* and *rightmost derivations* have the same parse tree.
- Grammar

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

- String

  id*id+id

- *leftmost derivations*: $E \Rightarrow E*E \Rightarrow \mathbf{id}*E \Rightarrow \mathbf{id}*E+E \Rightarrow \mathbf{id}*\mathbf{id}+E \Rightarrow \mathbf{id}*\mathbf{id}+\mathbf{id}$
- *rightmost derivations*: $E \Rightarrow E*E \Rightarrow E*E+E \Rightarrow E*E+\mathbf{id} \Rightarrow E*\mathbf{id}+\mathbf{id} \Rightarrow \mathbf{id}*\mathbf{id}+\mathbf{id}$

Parse Trees and Derivations

- Grammar

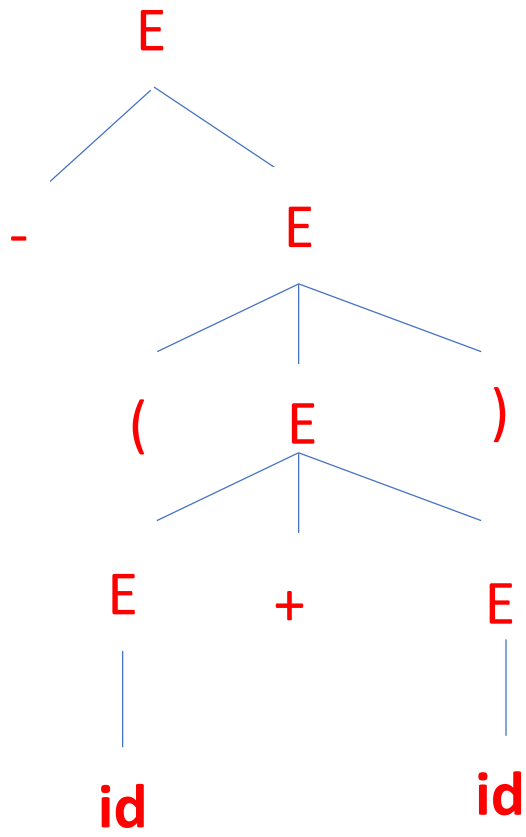
  E → E+E | E*E | -E | (E) | **id**

- String

  -(id+id)

- Derivation

  E
  ⇒ - E
  ⇒ - (E)
  ⇒ - (E+E)
  ⇒ - (**id**+E)
  ⇒ - (**id+id**)



Parse Trees and Derivations

E

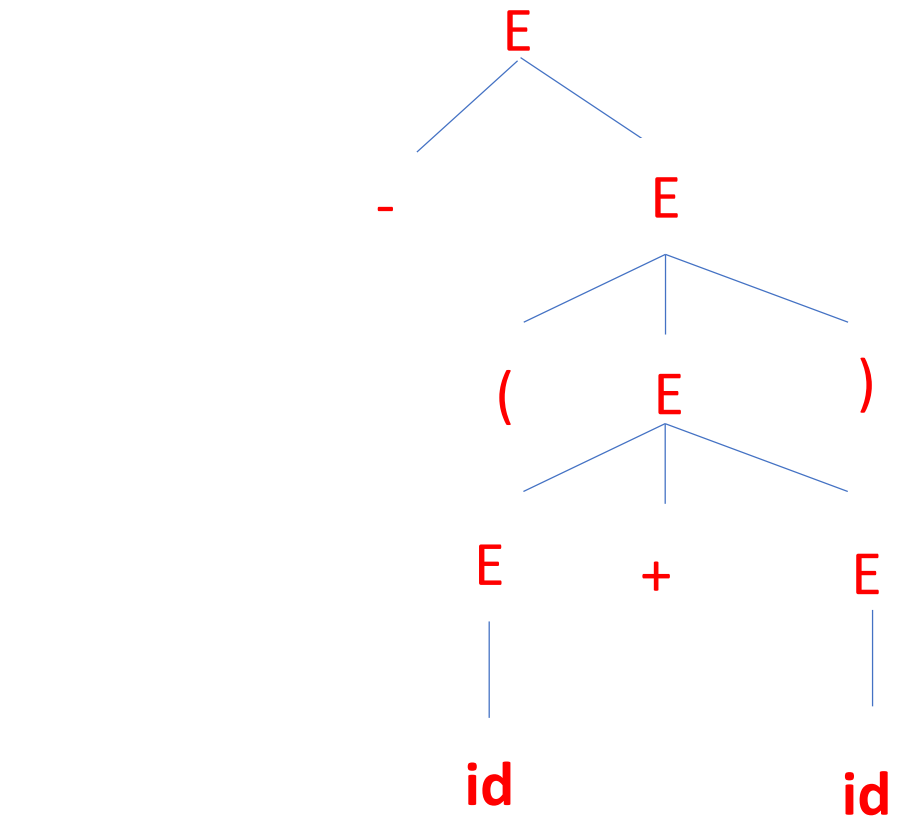
⇒ - E

⇒ - (E)

⇒ - (E+E)

⇒ - (**id**+E)

⇒ - (**id**+**id**)



Ambiguity

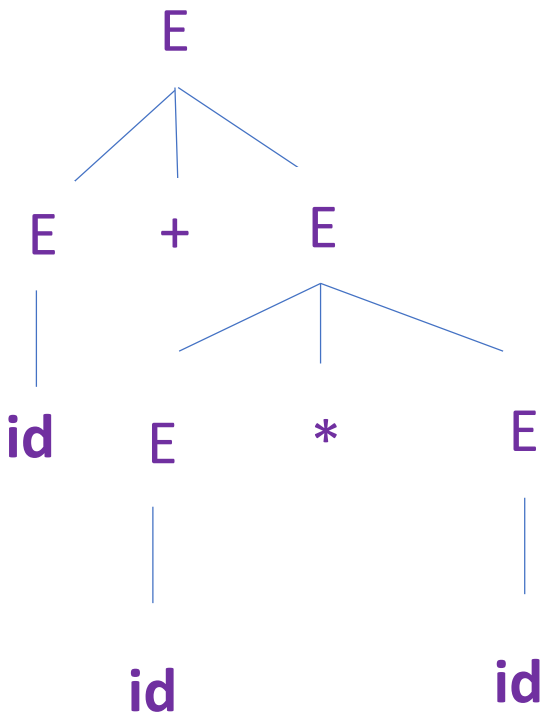
- A grammar that produces more than one parse tree for some string is said to be ambiguous.
  - Equivalently, there is more than one *leftmost derivations* or *rightmost derivations* for some string.
- Grammar

  $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid \mathbf{id}$
- String

  id+id*id
- This string has two parse trees

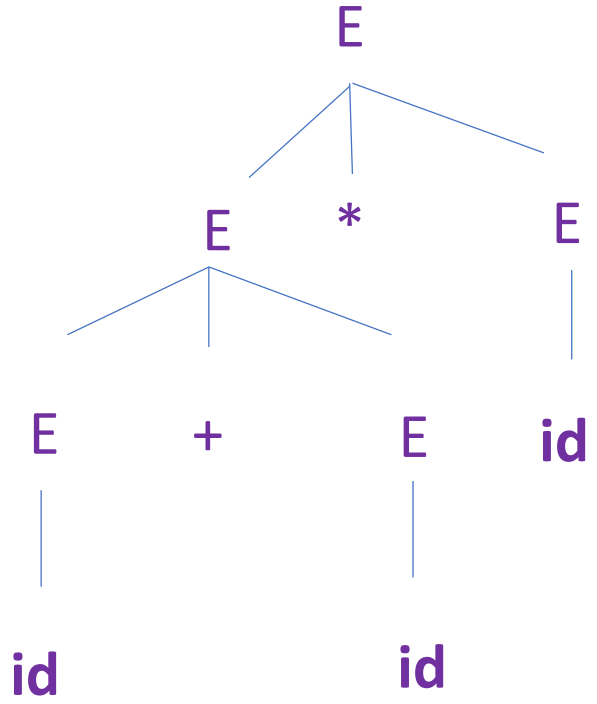
Ambiguity

- This string (**id**+**id***id**) has two parse trees



E ⇒ E+ E
⇒ **id**+E
⇒ **id**+E*E
⇒ **id**+**id***E
⇒ **id**+**id***id

E ⇒ E*E
⇒ E+E*E
⇒ **id**+E*E
⇒ **id**+**id***E
⇒ **id**+**id***id



Exercises

Exercise 1: Consider the context-free grammar:

$$S \rightarrow SS+ \mid SS* \mid a$$

and the string aa+a*

a) Give a derivation for the string
b) Give a leftmost derivation for the string
c) Give a rightmost derivation for the string
d) Give a parse tree for the string
e) Is the grammar ambiguous or unambiguous? Justify your answer
f) Describe the language generated by this grammar

Exercises

Exercise 2: Repeat Exercise 1 for each of the following grammars and strings:

a) $S \rightarrow 0S1 \mid 01$ with string 000111

b) $S \rightarrow +SS \mid *SS \mid a$ with string +*aaa

c) $S \rightarrow S(S)S \mid \varepsilon$ with string (()())

d) $S \rightarrow S+S \mid SS \mid (S) \mid S^* \mid a$ with string (a+a)*a

e) $S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$ with string ((a,a),a,(a))

f) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$ with string aabbab

Exercises

Exercise 3: Design grammars for the following languages:

a) The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backward as forward.

b) The set of all strings of 0s and 1s with an equal number of 0s and 1s

Exercises

Exercise 4:

- Let Σ = {**void**, **int**, **float**, **double**, **name**, (,), ‘,’, ;}.
- Examples:

```
void name(int name, double name);
int name();
int name(double name);
int name(int, int name, float);
void name(void);
```

Exercises

Here's one possible grammar:

S → *ret* **name** (*args*);

ret → *type* | **void**

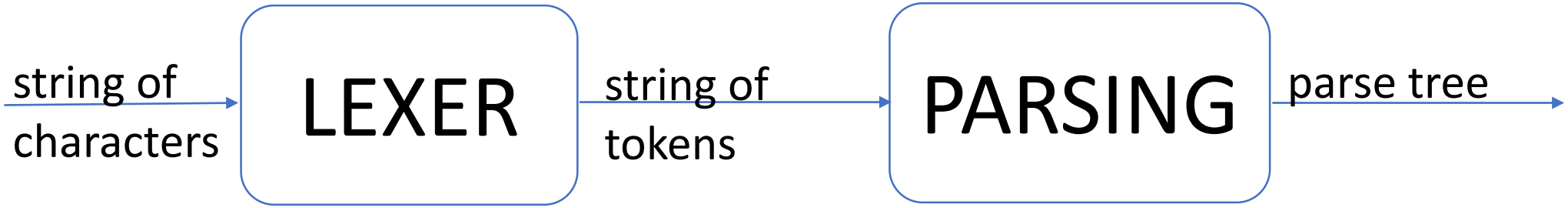
type → **int** | **double** | **float**

args → ε | **void** | *arglist*

arglist → *onearg* | *arglist*, *onearg*

onearg → *type* | *type* **name**

Abstract Syntax Trees (AST)



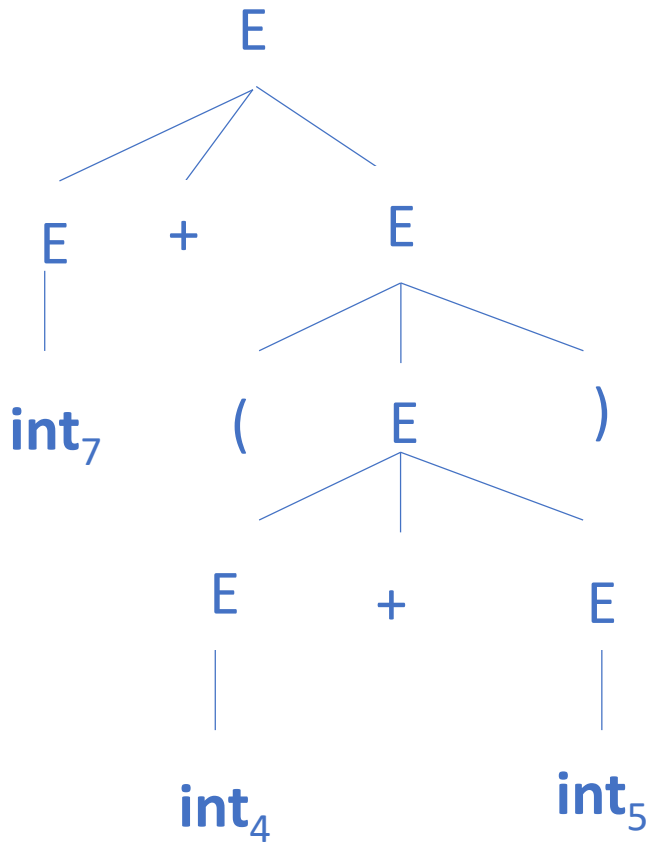
- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
  - Parse tree (Concrete Syntax Tree – CST)
- Abstract syntax trees like parse trees but ignore some details
  - In the abstract syntax tree (syntax tree), interior nodes represent programming constructs
  - In the parse tree (concrete syntax trees), interior nodes represent nonteminals
- Abstract Syntax Trees or ASTs are tree representations of code.

Abstract Syntax Trees (AST)

- Consider the grammar: E → **int** | (E) | E+E | E*E
- And the string: 7 + (4 + 5)
- After lexical analysis: **int**$_7$ '+' '(' **int**$_4$ '+' **int**$_5$ ')'

Example of parse tree

- During parsing we build a parse tree

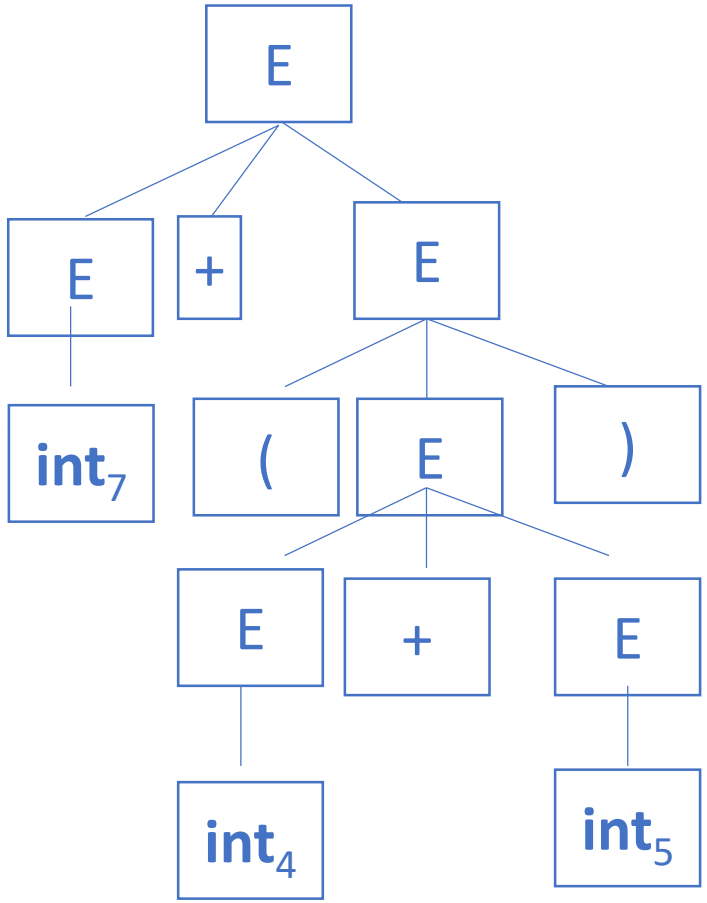


Too much information

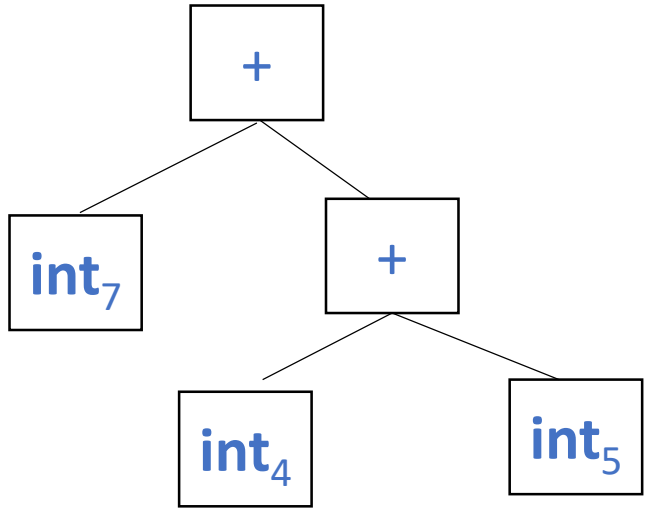
- Parentheses
- Single-successor nodes

Example of abstract syntax tree

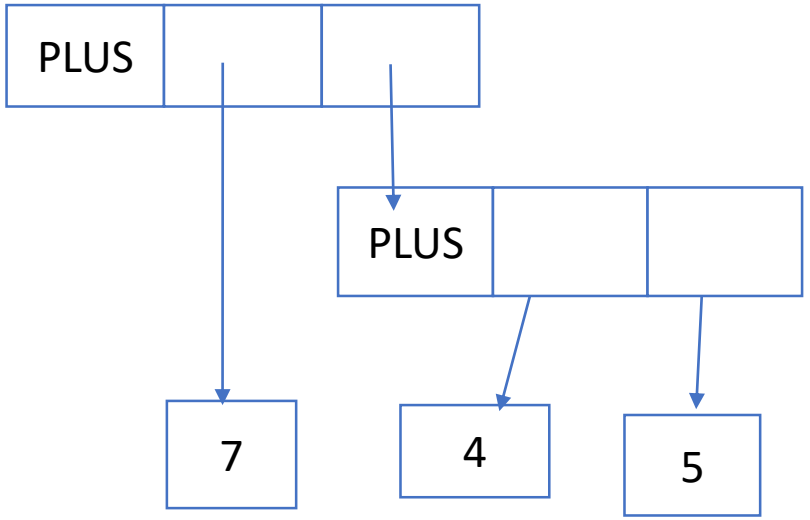
Parse Tree



Abstract Syntax Tree



AST is represented as



Parse Tree vs Syntax Tree

Parse Tree	Syntax Tree
A parse tree is a graphical representation of the replacement process in a derivation	A syntax tree is a condensed form of parse tree
In parse trees - Each interior node represents a grammar rule - Each leaf node represents a terminal	In syntax trees - Each interior node represents an operator - Each leaf node represents an operand
Parse tree represent every detail from the real syntax	Syntax trees does not represent every detail from the real syntax (that's why they are called abstract). For example: no rule nodes, no parentheses, …

Parse Tree vs Syntax Tree

- Consider the following grammar:

  $E \rightarrow E+T \mid T$

  $T \rightarrow T*F \mid F$

  $F \rightarrow (E) \mid \mathbf{id}$

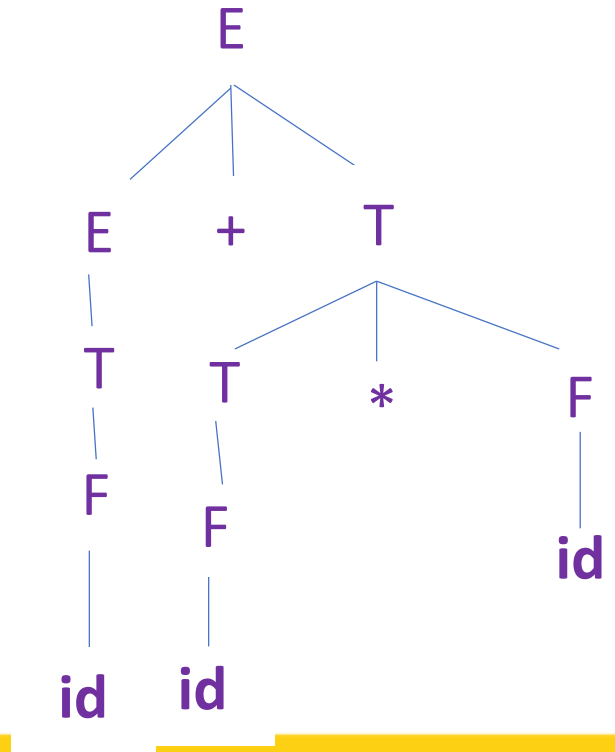
- For the string

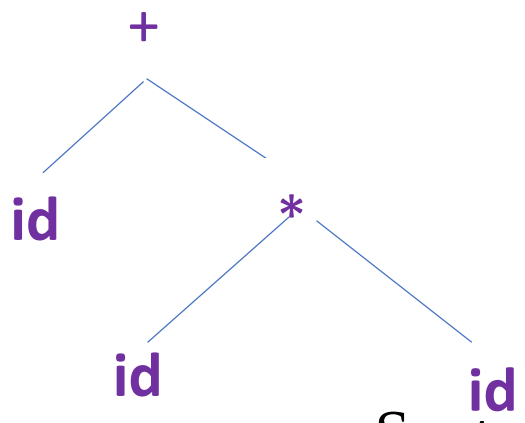
  id+id*id

- Genarate

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

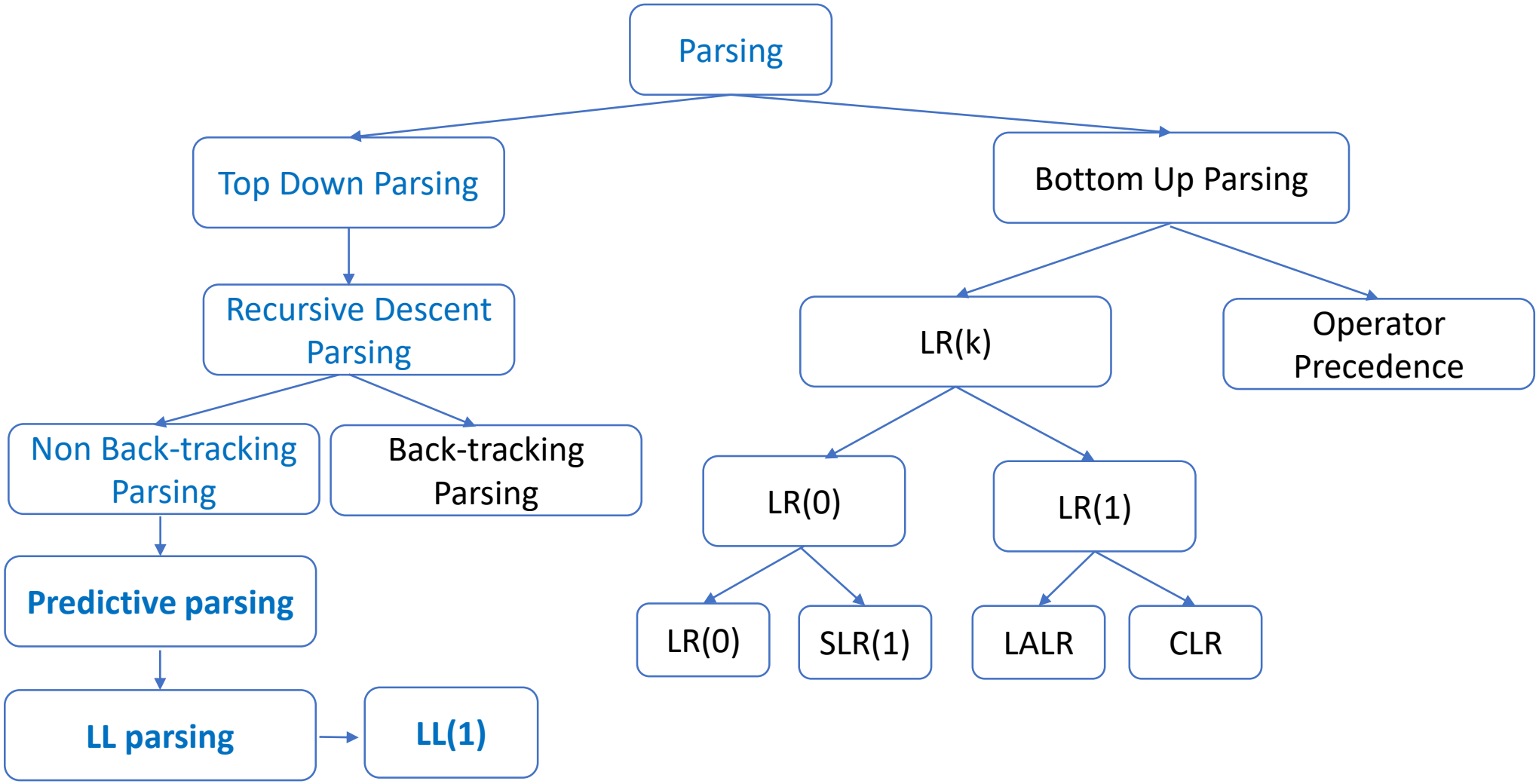
Parse tree





Syntax tree

Parsing


Parsing
Top Down Parsing
Bottom Up Parsing
Recursive Descent Parsing
LR(k)
Operator Precedence
Non Back-tracking Parsing
Back-tracking Parsing
LR(0)
LR(1)
Predictive parsing
LR(0)
SLR(1)
LALR
CLR
LL parsing
LL(1)

Top-Down Parsing

- Parsing is a process of deriving string from a given grammar.
- Top-down parsing is a process of constructing a parse tree for the input string:
  - From the top
  - From left to right
- Top-down parsing can be viewed as finding a leftmost derivation for an input string
- Recursive Descent Parsing uses procedures for every terminal and nonterminal entity.
- Recursive Descent Parsing to make a parse tree, which may or may not require back-tracking
- A form of recursive descent parsing that does not require any back-tracking is known as **predictive parsing**.

Top-Down Parsing

- Consider the grammar:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

and the input string: **id+id*id**

- Give a leftmost derivation for the input string
- Give a parse tree for the input string

Top-Down Parsing

- Consider the grammar:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

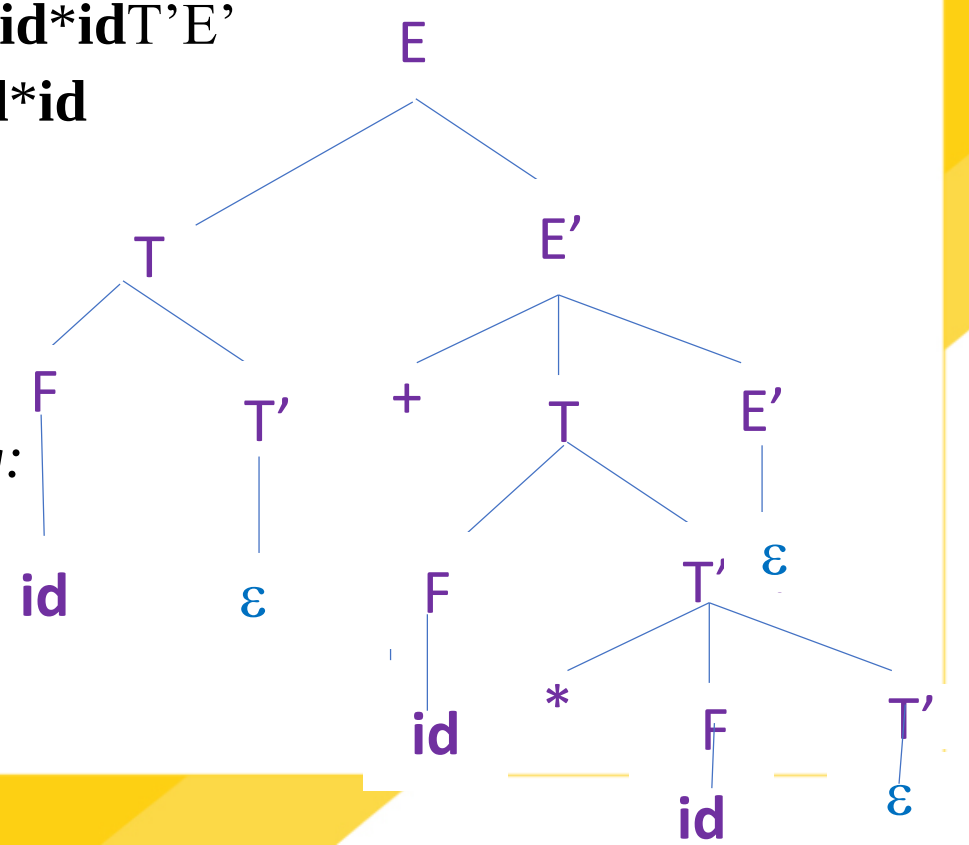
$F \rightarrow (E) \mid \mathbf{id}$

and the string: **id+id*id**

- *Give a leftmost derivation for the string:*

$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \mathbf{id}T'E' \Rightarrow \mathbf{id}E'$
$\Rightarrow \mathbf{id}+TE' \Rightarrow \mathbf{id}+FT'E' \Rightarrow \mathbf{id}+\mathbf{id}T'E'$
$\Rightarrow \mathbf{id}+\mathbf{id}*FT'E' \Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}T'E'$
$\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}E' \Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}$

Give a parse tree for the string:



Left-recursion

- A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

  $A \Rightarrow^{+} A\alpha$

  where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

- Recursive descent parsing does not work in such cases.

Elimination of Left-Recursion

- Consider the left-recursive grammar $A \rightarrow A\alpha \mid \beta$
  where α and β are sequences of terminals and nonterminals that do not start with A.

- The nonterminal A and its production are said to be left-recursive, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right hand side.

- A generates all strings starting with a β and followed by a number of α.

- For example, in *expr* → *expr*+*term* | *term*
  nonterminal A = *expr*, string α = +*term*, and string β = *term*

expr⇒*expr*+*term*⇒*expr*+*term*+*term*⇒*expr*+*term*+*term*+*term*⇒*expr*+*term*+*term*+*term*+*term*

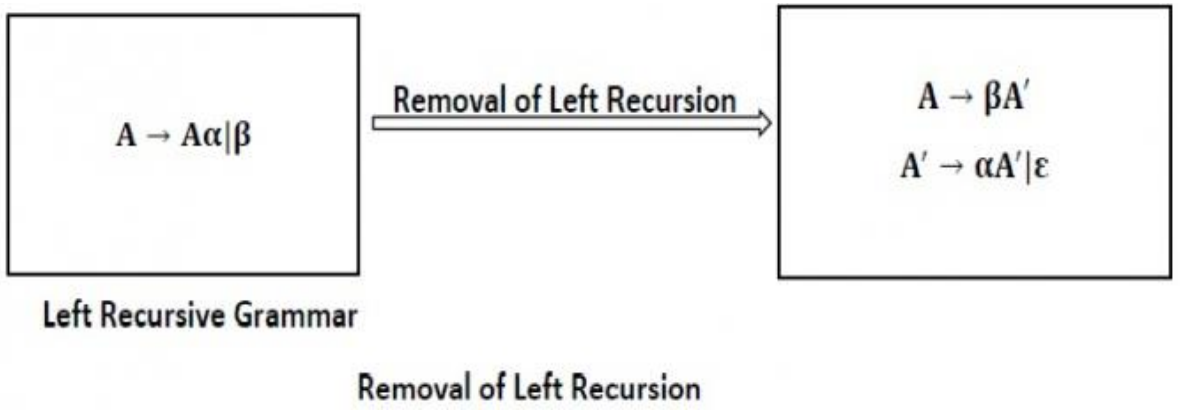
Removing left-recursion

- Left-recursion often poses problems for parsers
- Therefore, a grammar is often preprocessed to eliminate the left-recursion.
- The left-recursive grammar $A \rightarrow A\alpha \mid \beta$

  can rewrite using right-recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$



Removing left-recursion

- **Example:**

Consider the left-recursive grammar

$S \rightarrow a \mid \hat{}(T)$

$T \rightarrow T,S \mid S$

Eliminate the left-recursion from the grammar.

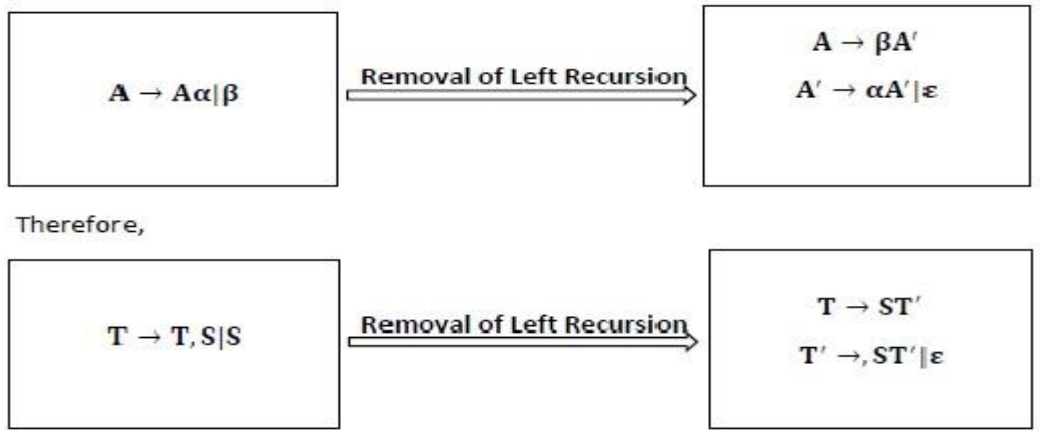
Solution: We have immediate left recursion in T-productions.

- Comparing $T \rightarrow T,S \mid S$ with $A \rightarrow A\alpha \mid \beta$ where A = T, α =, S and β = S
- Complete grammar will be

$S \rightarrow a \mid \hat{}(T)$

$T \rightarrow ST'$

$T' \rightarrow ,ST' \mid \varepsilon$



Removing left-recursion

Solution

- Comparing $E \rightarrow E+T \mid T$ with $A \rightarrow A\alpha \mid \beta$
  - $A = E, \alpha = +T, \beta = T$
  - 1) $A \rightarrow A\alpha \mid \beta$ is changed to $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$
  - 2) $A \rightarrow \beta A'$ means $E \rightarrow TE'$
  - 3) $A' \rightarrow \alpha A' \mid \varepsilon$ means $E' \rightarrow +TE' \mid \varepsilon$
- Comparing $T \rightarrow T*F \mid F$ with $A \rightarrow A\alpha \mid \beta$
  - 4) $A = T, \alpha = *F, \beta = F$
  - 5) $A \rightarrow \beta A'$ means $T \rightarrow FT'$
  - 6) $A \rightarrow \alpha A' \mid \varepsilon$ means $T' \rightarrow *FT' \mid \varepsilon$
- Production $F \rightarrow (E) \mid id$ does not have any left-recursion
- Combining productions 1, 2, 3, 4, 5, 6 we get

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

  $F \rightarrow (E) \mid \mathbf{id}$

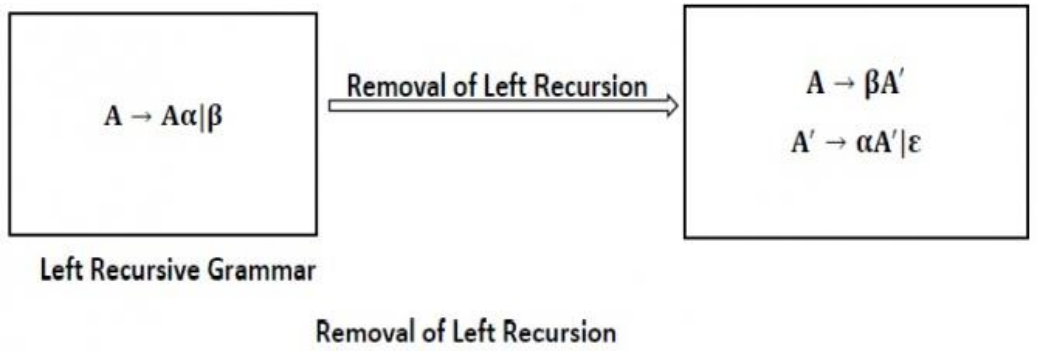
Example:

Consider the left-recursive grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$



Removing left-recursion

Example:

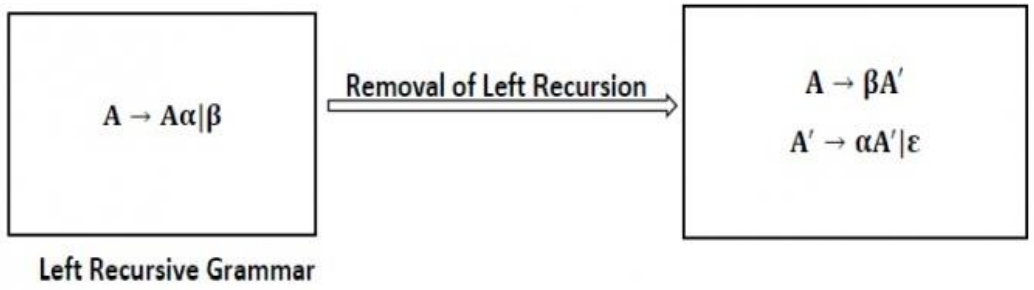
- The left-recursive grammar $A \rightarrow A\alpha \mid \varepsilon$ can rewrite using right-recursion

$A \rightarrow A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

or

$A \rightarrow \alpha A \mid \varepsilon$



Removing left-recursion

Example:

- Consider the left-recursive grammar:

$S \rightarrow SabA \mid a$

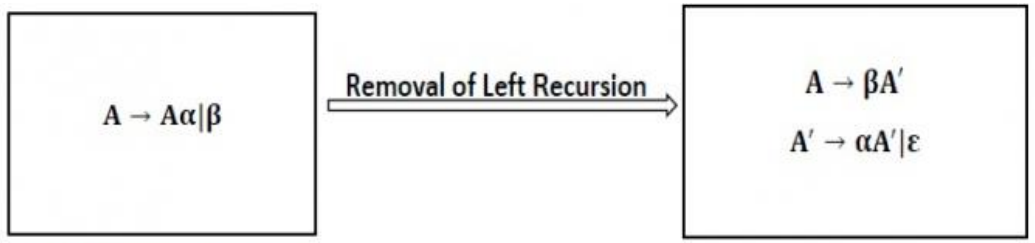
$A \rightarrow a \mid b$

- Can rewrite as

$S \rightarrow aS'$

$S' \rightarrow abAS' \mid \varepsilon$

$A \rightarrow a \mid b$



Removing left-recursion

- In general $A \rightarrow A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$

- A generates all strings starting with one of $\beta_1, \ldots, \beta_m$ and followed by several instances of $\alpha_1, \ldots, \alpha_n$

- Can rewrite as

$A \rightarrow \beta_1 A' | \ldots | \beta_m A'$
$A' \rightarrow \alpha_1 A' | \ldots | \alpha_n A' | \varepsilon$

Removing left-recursion

Example:

- Consider the left-recursive grammar:

$S \rightarrow SbA \mid SaA \mid b \mid a$

$A \rightarrow a \mid b$

- Can rewrite as

$S \rightarrow bS' \mid aS'$

$S' \rightarrow bAS' \mid aAS' \mid \varepsilon$

$A \rightarrow a \mid b$

General Left-Recursion

- The grammar

  $A \rightarrow A'\alpha \mid \beta$

  $A' \rightarrow A\delta$

➢Is it a left-recursive grammar?

- It is also left-recursive because:

$A \Rightarrow A'\alpha \Rightarrow A\delta\alpha$ or $A \Rightarrow^{+} A\delta\alpha$

A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

$A \Rightarrow^{+} A\alpha,$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

General Left-Recursion

- Consider the grammar
  $S \rightarrow Aa \mid b$
  $A \rightarrow Ac \mid Sd \mid \varepsilon$

➢The nonterminal S is left-recursive?

A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

$A \Rightarrow^{+} A\alpha,$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

- The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$ or $S \Rightarrow^{+} Sda$, but it is not immediately left-recursive.

Left-Recursion

- Consider the grammar

  $S \rightarrow a \mid \wedge(T)$

  $T \rightarrow ST'$

  $T' \rightarrow ,ST' \mid \varepsilon$

➢ Is it a left-recursive grammar?

A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

$A \Rightarrow^{+} A\alpha,$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

$S \Rightarrow \wedge(T) \Rightarrow \wedge(ST') \Rightarrow \wedge(\wedge(T)T') \Rightarrow \ldots\ldots \Rightarrow \wedge\alpha$

$T \Rightarrow ST' \Rightarrow \ldots\ldots \Rightarrow \wedge\,\alpha$

$T' \Rightarrow ,ST' \Rightarrow \ldots\ldots \Rightarrow ,\wedge\,\alpha$

Algorithm For Eliminating Left-Recursion

- INPUT: Grammar G with no cycles (derivations of the form $A \Rightarrow^{+} A$) or ε-productions.
- OUTPUT: An equivalent grammar with no left-recursion.
  (Note that the resulting non-left-recursive grammar may have ε-productions)
- METHOD:

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) for (each i from 1 to n) {
3) for (each j from 1 to i-1) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_i-productions
5) }
6) eliminate the immediate left-recursion among the A_i-productions
7) }

Example

- Let apply that algorithm (Algorithm For Eliminating Left-Recursion in the previous slide) to the grammar

  $A \rightarrow B\alpha \mid \beta$

  $B \rightarrow A\delta$

Solution:

- We order the nonterminals: A, B
- For i = 1: nothing happens (because there is no immediate left-recursion among the A-productions)
- For i = 2:
  - For j = 1: replace the production $B \rightarrow A\delta$ by two productions:

    $B \rightarrow B\alpha\delta$ (where $A \rightarrow B\alpha$)

    and $B \rightarrow \beta\delta$ (where $A \rightarrow \beta$)

    $A \rightarrow B\alpha \mid \beta$

    $B \rightarrow B\alpha\delta \mid \beta\delta$
  - eliminate immediate left-recursion:

    $A \rightarrow B\alpha \mid \beta$

    $B \rightarrow \beta\delta B'$

    $B' \rightarrow \alpha\delta B' \mid \varepsilon$

Exercise

1. Eliminate the left-recursion from the grammar:

$A \rightarrow AT \mid T$

$T \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

2. Eliminate the left-recursion from the grammar:

$A \rightarrow AB \mid AC \mid C$

$C \rightarrow a$

$B \rightarrow 0$

3. Eliminate the left-recursion from the grammar:

$A \rightarrow B\alpha \mid \beta$

$B \rightarrow A\delta \mid \beta \mid cd$

$C \rightarrow db \mid bc$

4. Eliminate the left-recursion from the grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd$

Exercise

1. Eliminate the left-recursion from the grammar:

$A \rightarrow AT \mid T$

$T \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Solution:

$A \rightarrow TA'$

$A' \rightarrow TA' \mid \varepsilon$

$T \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

2. Eliminate the left-recursion from the grammar:

$A \rightarrow AB \mid AC \mid C$

$C \rightarrow a$

$B \rightarrow 0$

Solution:

$A \rightarrow CA'$

$A' \rightarrow BA' \mid CA' \mid \varepsilon$

$C \rightarrow a$

$B \rightarrow 0$

Exercise

3. Eliminate the left-recursion from the grammar

$A \rightarrow B\alpha \mid \beta$

$B \rightarrow A\delta \mid A\beta \mid cd$

$C \rightarrow db \mid bc$

- Solution:

- We order the nonterminals: A, B, C.
- For i = 1: nothing happens (because there is no immediate left recursion among the A-productions)
- For i = 2:
  - For j = 1: - substitute for A in $B \rightarrow A\delta$ to obtain the following B-productions: $B \rightarrow B\alpha\delta \mid \beta\delta$
    - substitute for A in $B \rightarrow A\beta$ to obtain the following B-productions: $B \rightarrow B\alpha\beta \mid \beta\beta$
    - we obtain:

$A \rightarrow B\alpha \mid \beta$

$B \rightarrow B\alpha\delta \mid B\alpha\beta \mid \beta\delta \mid \beta\beta \mid cd$

$C \rightarrow db \mid bc$

- eliminate the immediate left recursion among these B-productions yields the following grammar

$A \rightarrow B\alpha \mid \beta$

$B \rightarrow \beta\delta B' \mid \beta\beta B' \mid cdB'$

$B' \rightarrow \alpha\delta B' \mid \alpha\beta B' \mid \varepsilon$

$C \rightarrow db \mid bc$

- For i = 3: nothing happens (because there is no immediate left recursion among the C-productions)

Left factoring

- Consider the grammar

  stmt → **if** *expr* **then** *stmt* **else** *stmt*

  | **if** *expr* **then** *stmt*

➢ on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*.

➢ we need to left-factor the grammar

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.
- Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
  and the input begins with a nonempty string derived from α

➢ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$

➢ we may defer the decision by

- expanding A to $\alpha A'$
- then exapanding A' to β_1 or β_2

➢ the original productions become

$A \rightarrow \alpha A'$

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

Left factoring

Example:

- Consider the grammar:

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

➢ Left-factor this grammar.

- Can rewrite as

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
and the input begins with a nonempty string derived from α

- ➢ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
- ➢ we may defer the decision by
  - expanding A to $\alpha A'$
  - then exapanding A' to β_1 or β_2
- ➢ the original productions become

$A \rightarrow \alpha A'$

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

Left factoring

- Consider the grammar
  $E \rightarrow T+E \mid T$
  $T \rightarrow \mathbf{int} \mid \mathbf{int}*T \mid (E)$

➢ Left-factor this grammar.

$E \rightarrow TE'$

$E' \rightarrow +E \mid \varepsilon$

$T \rightarrow \mathbf{int}T' \mid (E)$

$T' \rightarrow *T \mid \varepsilon$

- Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$ and the input begins with a nonempty string derived from α
- ➢ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
- ➢ we may defer the decision by
  - expanding A to $\alpha A'$
  - then exapanding A' to β_1 or β_2
- ➢ the original productions become
  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Predictive Parsing

- Recursive-descent parsing is a top-down method of syntax analysis
  - in which a set of recursive procedures is used to process the input.
  - one procedure is associated with each nonterminal of a grammar.
- Predictive parsing is a special case of recursive-descent parsing, but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k)
  - **L** means "left-to-right" scan of input
  - **L** means "leftmost derivation"
  - **k** means "predict based on k tokens of lookahead"
  - In practice, **LL(1)** is used

LL(1) grammars

- Predictive parsers
  - is recursive-descent parsers needing no backtracking
  - can be constructed for a class of grammars called LL(1)
- LL(1)
  - First L stands for scanning the input from left to right
  - Second L stands for producing a leftmost derivation
  - 1 stands for using one input symbol of lookahead at each step to make parsing action decisions.
- LL(1) grammars are not ambiguous and not left recursive.

LL(1) vs. Recursive Descent

- In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1),
  - At each step, only one choice of production
  - That is
    - When a nonterminal A is leftmost in a derivation
    - And the next input symbol is t
    - There is a unique production $A \rightarrow \alpha$ to use (or no production to use (an error state))
- LL(1) is a recursive descent variant without backtracking

FIRST and FOLLOW

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input token.
- FIRST
  - Definition: FIRST(X) to be the set of terminals that begin strings derived from X
    FIRST(X) = {b | $X \Rightarrow^+ b\alpha$}, $\varepsilon \in$ FIRST(X) if $X \Rightarrow^+ \varepsilon$
  - Compute FIRST(X):
    - If X is a terminal: FIRST(X) = {X}
    - If X is a nonterminal:
      - $\varepsilon \in$ FIRST(X)
        - If $X \rightarrow \varepsilon$
        - If $X \rightarrow Y_1 Y_2 \ldots Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all $1 \leq i \leq k$
      - $b \in$ FIRST(X)
        - If $X \rightarrow Y_1 Y_2 \ldots Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \leq j \leq i-1$

FIRST example

- Production Rules of Grammar

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

  $F \rightarrow (E) \mid \mathbf{id}$
- Compute FIRST

FIRST example

- Production Rules of Grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

FIRST (+) = {+}
FIRST (*) ={*}
FIRST (() = {(}
FIRST ()) ={)}
FIRST (**id**) ={**id**}

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

Compute FIRST(X):
If X is a terminal: FIRST(X) = {X}
If X is a nonterminal:
ε ∈ FIRST(X):
If $X \rightarrow \varepsilon$
If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in FIRST(Y_i)$ for all $1 \leq i \leq k$
b ∈ FIRST(X):
If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$ for all $1 \leq j \leq i-1$

FIRST example

- Consider the grammar

  $E \rightarrow TF$

  $F \rightarrow +E \mid \varepsilon$

  $T \rightarrow \textbf{int}Y \mid (E)$

  $Y \rightarrow *T \mid \varepsilon$

- Compute FIRST

FIRST example

- Consider the grammar

$E \rightarrow TF$

$F \rightarrow +E \mid \varepsilon$

$T \rightarrow \textbf{int}Y \mid (E)$

$Y \rightarrow *T \mid \varepsilon$

FIRST (+) = {+}
FIRST (*) ={*}
FIRST (() = {(}
FIRST ()) ={)}
FIRST (**int**) = {**int**}

FIRST (T) ={**int, (**}
FIRST (E) ={**int, (**}
FIRST (F) ={+, ε}
FIRST (Y) ={*, ε}

Compute FIRST(X):
If X is a terminal: FIRST(X) = {X}
If X is a nonterminal:
ε ∈ FIRST(X):
If $X \rightarrow \varepsilon$
If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in FIRST(Y_i)$ for all $1 \leq i \leq k$
b ∈ FIRST(X):
If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$ for all $1 \leq j \leq i-1$

FOLLOW example

- FOLLOW
  - Definition: FOLLOW (X) to be the set of terminals b that can appear immediately to the right of nonterminal X in some sentential form.

    FOLLOW (X) = $\{b \mid S \Rightarrow^{+} \beta X b \delta\}$, where X is a nonterminal
  - Compute FOLLOW(X):
    - $ ∈ FOLLOW(S), where S is the start symbol, $ is the end symbol of the input string
    - For each production $A \rightarrow \alpha X \beta$: $FIRST(\beta) \backslash \{\varepsilon\} \subseteq FOLLOW(X)$
    - For each production $A \rightarrow \alpha X \beta$, where $\varepsilon \in FIRST(\beta)$: $FOLLOW(A) \subseteq FOLLOW(X)$
    - For each production $A \rightarrow \alpha X$: $FOLLOW(A) \subseteq FOLLOW(X)$

FOLLOW example

- Production Rules of Grammar

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

  $F \rightarrow (E) \mid \mathbf{id}$

- Compute FOLLOW

FOLLOW example

- Production Rules of Grammar

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

  $F \rightarrow (E) \mid \mathbf{id}$

Compute FOLLOW(X):

$ ∈ FOLLOW(S), where S is the start symbol, $ is the end symbol of the input string

For each production $A \rightarrow \alpha X \beta$: $FIRST(\beta) \setminus \{\varepsilon\} \subseteq FOLLOW(X)$

For each production $A \rightarrow \alpha X \beta$, where $\varepsilon \in FIRST(\beta)$: $FOLLOW(A) \subseteq FOLLOW(X)$

For each production $A \rightarrow \alpha X$: $FOLLOW(A) \subseteq FOLLOW(X)$

FIRST (+) = {+}
FIRST (*) ={*}
FIRST (() = {(}
FIRST ()) ={)}
FIRST (**id**) ={**id**}

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

FOLLOW(E)={$,)}
FOLLOW(T)={+, $,)}
FOLLOW(E')={$,)}
FOLLOW(F)={*, +, $,)}
FOLLOW(T')={+, $,)}

FOLLOW example

- Recall the grammar

  $E \rightarrow TF$

  $F \rightarrow +E \mid \varepsilon$

  $T \rightarrow \textbf{int}Y \mid (E)$

  $Y \rightarrow *T \mid \varepsilon$

- Compute FOLLOW

FOLLOW example

- Recall the grammar

  $E \rightarrow TF$

  $F \rightarrow +E \mid \varepsilon$

  $T \rightarrow \textbf{int}Y \mid (E)$

  $Y \rightarrow *T \mid \varepsilon$

Compute FOLLOW(X):

$\$ \in FOLLOW(S)$, where S is the start symbol, $ is the end symbol of the input string

For each production $A \rightarrow \alpha X \beta$: $FIRST(\beta) \setminus \{\varepsilon\} \subseteq FOLLOW(X)$

For each production $A \rightarrow \alpha X \beta$, where $\varepsilon \in FIRST(\beta)$: $FOLLOW(A) \subseteq FOLLOW(X)$

For each production $A \rightarrow \alpha X$: $FOLLOW(A) \subseteq FOLLOW(X)$

FIRST (+) = {+}

FIRST (*) ={*}

FIRST (() = {(}

FIRST ()) ={)}

FIRST (**int**) = {**int**}

FIRST (T) ={**int**, (}

FIRST (E) ={**int**, (}

FIRST (F) ={+, ε}

FIRST (Y) ={*, ε}

FOLLOW(E) = {$,)}

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

FOLLOW(F) ={$,)}

FOLLOW(Y) ={+, $,)}

FOLLOW example

- Consider the grammar
  stmt → *ifst* | **other**
  ifst → **if** (*exp*) *stmt elsepart*
  elsepart → **else** *stmt* | ε
  exp → 0 | 1
- Compute FIRST, FOLLOW

FOLLOW set example

- Consider the grammar
  stmt → *ifst* | **other**
  ifst → **if** (*exp*) *stmt elsepart*
  elsepart → **else** *stmt* | ε
  exp → 0 | 1

FIRST(**other**) ={**other**}
FIRST(**if**) ={**if**}
FIRST(**else**) ={**else**}
FIRST(() ={(}
FIRST()) ={)}
FIRST(0) ={0}
FIRST(1) ={1}

FIRST(*stmt*) ={**other, if**}
FIRST(*ifst*) ={**if**}
FIRST(*exp*) = {0,1}
FIRST(*elsepart*) ={ε, **else**}

Compute FIRST(X):
If X is a terminal: FIRST(X) = {X}
If X is a nonterminal:
ε ∈ FIRST(X):
If $X \rightarrow \varepsilon$
If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in FIRST(Y_i)$ for all $1 \le i \le k$
b ∈ FIRST(X):
If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$ for all $1 \le j \le i-1$

Compute FOLLOW(X):
$ ∈ FOLLOW(S), where S is the start symbol, $ is the end symbol of the input string
For each production $A \rightarrow \alpha X\beta$: $FIRST(\beta)\backslash\{\varepsilon\} \subseteq FOLLOW(X)$
For each production $A \rightarrow \alpha X\beta$, where $\varepsilon \in FIRST(\beta)$: $FOLLOW(A) \subseteq FOLLOW(X)$
For each production $A \rightarrow \alpha X$: $FOLLOW(A) \subseteq FOLLOW(X)$

FOLLOW(*stmt*) = {$, **else**}
FOLLOW(*ifst*) ={$, **else**}
FOLLOW(*elsepart*) ={$, **else**}
FOLLOW(*exp*) = {)}

LL(1) Parsing Table

- Construct a parsing table M for context free grammar G.
  - M is a two-dimensional array M[A,t], where A is a nonterminal and t is a terminal or $ (the end symbol of the input string)

NON-TERMINAL	INPUT SYMBOL					
		t				$
A		α				α

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $t \in FIRST(\alpha)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do $M[A, \$] = \alpha$

LL(1) Parsing Table

- Given left-factored grammar

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

  $F \rightarrow (E) \mid \mathbf{id}$

FIRST (+) = {+}
FIRST (*) ={*}
FIRST (() = {(}
FIRST ()) ={)}
FIRST (**id**) ={**id**}

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

FOLLOW(E)={$,)}
FOLLOW(T)={+, $,)}
FOLLOW(E')={$,)}
FOLLOW(F)={*, +, $,)}
FOLLOW(T')={+, $,)}

FIRST(TE')= FIRST(T)={(, **id**}
FIRST(+TE')= FIRST(+)={+}
FIRST(FT')= FIRST(F)={(, **id**}
FIRST(*FT')= FIRST(*)={*}
FIRST((E))= FIRST(()={(}
FIRST(ε)= {ε}

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do M[A, t] = α
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do M[A, t] = α
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do M[A, $] = α

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	**id**			(E)		

LL(1) Parsing Table

- Consider the grammar

  $E \rightarrow TF$

  $F \rightarrow +E \mid \varepsilon$

  $T \rightarrow \mathbf{int}Y \mid (E)$

  $Y \rightarrow *T \mid \varepsilon$

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do $M[A, \$] = \alpha$

FIRST (+) = {+}
FIRST (*) ={*}
FIRST (() = {(}
FIRST ()) ={)}
FIRST (**int**) = {**int**}

FIRST (T) ={**int, (**}
FIRST (E) ={**int, (**}
FIRST (F) ={+, ε}
FIRST (Y) ={*, ε}

FOLLOW(E) = {$,)}
FOLLOW(T) ={+, $,)}
FOLLOW(F) ={$,)}
FOLLOW(Y) ={+, $,)}

FIRST(TF)=FIRST(T)={**int, (**}
FIRST(+E)=FIRST(+)={+}
FIRST(**int** Y)=FIRST(**int**)={**int**}
FIRST((E))=FIRST(( )={(}
FIRST(*T)=FIRST(*)={*}
FIRST(ε)= {ε}

NON-TERMINAL	INPUT SYMBOL					
	+	**int**	(	)	*	$
E		TF	TF			
F	+E			ε		ε
T		**int**Y	(E)			
Y	ε			ε	*T	ε

LL(1) Parsing Table

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	**id**			(E)		

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | **id**

- Consider the [E, id] entry
  - “When current nonterminal is E and next input is **id**, use production E → TE’ “
  - This can generate an **id** in the first position

LL(1) Parsing Table

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	**id**			(E)		

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | **id**

- Consider the [T’, +] entry
  - “When current nonterminal is T’ and current token is +, get rid of T’ “
  - T’ can be followed by + only of T’ → ε

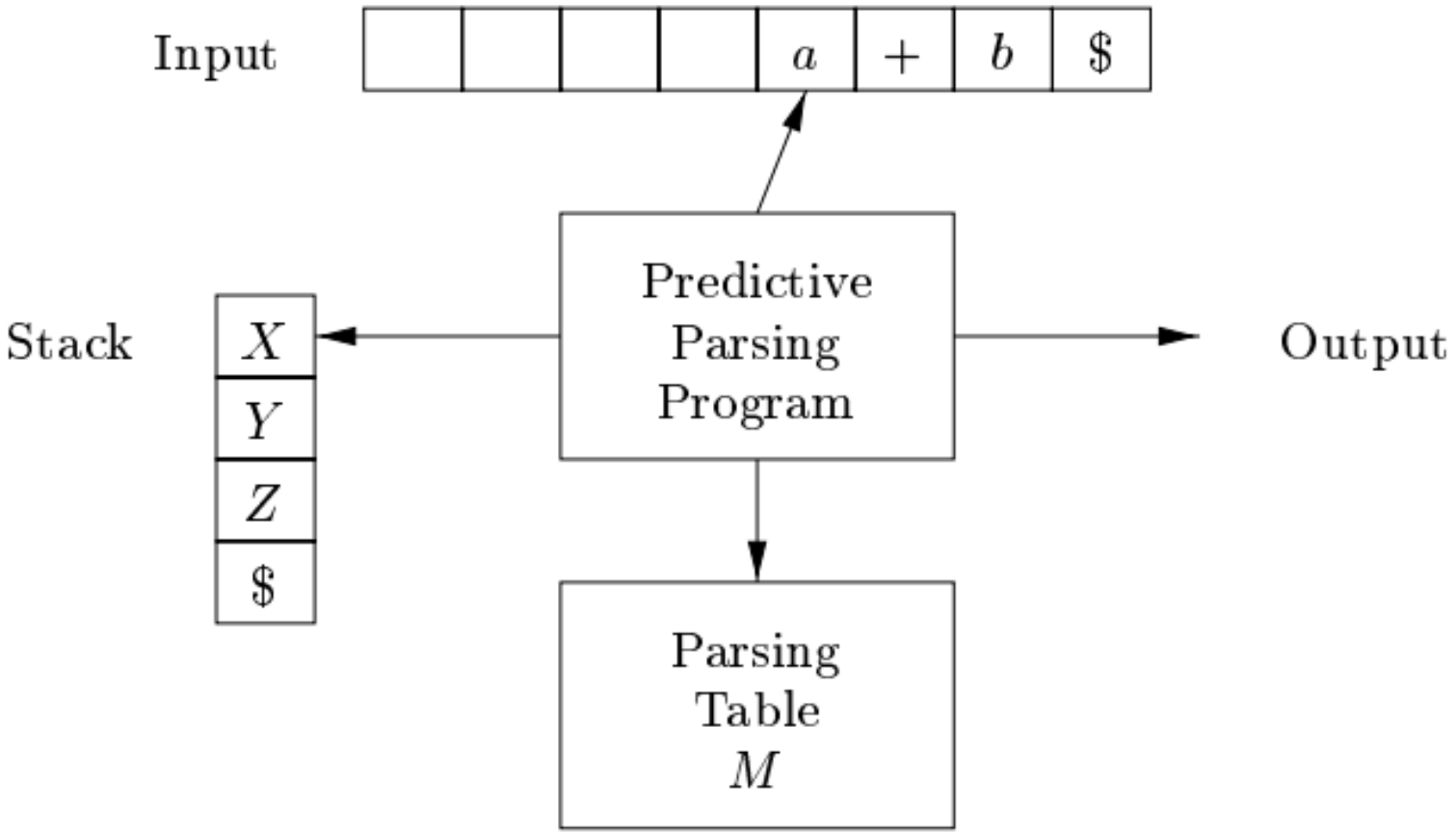
LL(1) Parsing Table

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	id			(E)		

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | **id**

- Consider the [F, *] entry
  - “There is no way to derive a string starting with * from nonterminal F “
  - Blank entries indicate error situations

Using Parsing Tables



Model of a table-driven predictive parser

Table-driven predictive parsing algorithm

- INPUT: a string w and a parsing table M for grammar G
- OUTPUT:
  - If w is in L(G): a leftmost derivation of w,
  - Otherwise, an error indication
- METHOD
  - the parser is in a cofiguration with w$ in the input buffer
  - and the start symbol S of G on top of the stack, above $
  - Follow program uses the predictive parsing table M to produce a predictive parse for the input

let a be the first symbol of w;
let X be the top stack symbol;
while (X ≠ $) /* stack is not empty */
{ **if** (X = a) pop the stack and **let** a be the next symbol of w;
else if (X is a terminal) *error*();
else if (M [X, a] is an error entry) *error*();
else if (M [X, a] = $Y_1Y_2\ldots Y_k$) { output the production $X \rightarrow Y_1Y_2\ldots Y_k$;
pop the stack;
push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
}
let X be the top stack symbol;
}

Example

- Given left-factored grammar
  E → TE’
  E’ → +TE’ | ε
  T → FT’
  T’ → *FT’ | ε
  F → (E) | **id**
- Input: **id+id*id**

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	**id**			(E)		

MATCHED	STACK	INPUT	ACTION
	E$	**id+id*id$**	
	TE’$	**id+id*id$**	Output E → TE’
	FT’E’$	**id+id*id$**	Output T → FT’
	idT’E’$	**id+id*id$**	Output F → **id**
id	T’E’$	**+id*id$**	Match **id**
id	E’$	**+id*id$**	Output T’ → ε
id	+TE’$	**+id*id$**	Output E’ → +TE’
id+	TE’$	**id*id$**	Match +
id+	FT’E’$	**id*id$**	Output T → FT’
id+	**id**T’E’$	**id*id$**	Output F → **id**
id+id	T’E’$	***id$**	Match **id**
id+id	*FT’E’$	***id$**	Output T’ → *FT’
id+id*	FT’E’$	**id$**	Match *
id+id*	**id**T’E’$	**id$**	Output F → **id**
id+id*id	T’E’$	$	Match **id**
id+id*id	E’$	$	Output T’ → ε
id+id*id	$	$	Output E’ → ε

Example

- Consider the grammar:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

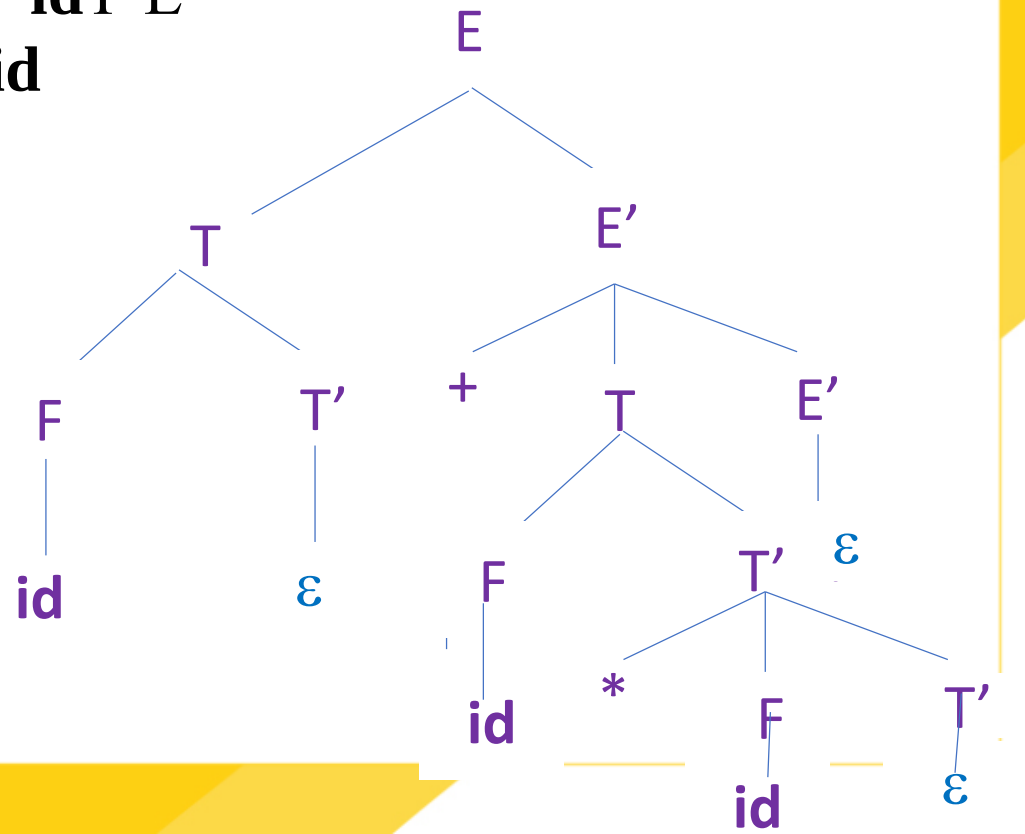
$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \mathbf{id}T'E' \Rightarrow \mathbf{id}E'$
$\Rightarrow \mathbf{id}+TE' \Rightarrow \mathbf{id}+FT'E' \Rightarrow \mathbf{id}+\mathbf{id}T'E'$
$\Rightarrow \mathbf{id}+\mathbf{id}*FT'E' \Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}T'E'$
$\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}E' \Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}$

- And the string: **id+id*id**
- Give a parse tree for the string



Example

- Consider the grammar
  $E \rightarrow TF$
  $F \rightarrow +E \mid \varepsilon$
  $T \rightarrow \mathbf{int}Y \mid (E)$
  $Y \rightarrow *T \mid \varepsilon$
- Input: **(int)**

NON-TERMINAL	INPUT SYMBOL					
	+	**int**	(	)	*	$
E		TF	TF			
F	+E			ε		ε
T		**int**Y	(E)			
Y	ε			ε	*T	ε

MATCHED	STACK	INPUT	ACTION
	E$	**(int)**$	
	TF$	**(int)**$	Output E → TF
	(E)F$	**(int)**$	Output T → (E)
(	E)F$	**int)**$	Match **(**
(	TF)F$	**int)**$	Output E → TF
(	**int**YF)F$	**int)**$	Output T → **int**Y
(int	YF)F$	**)**$	Match **int**
(int	F)F$	**)**$	Output Y → ε
(int	)F$	**)**$	Output F → ε
(int)	$	$	Output F → ε

Example

- Consider the grammar

  $E \rightarrow TF$

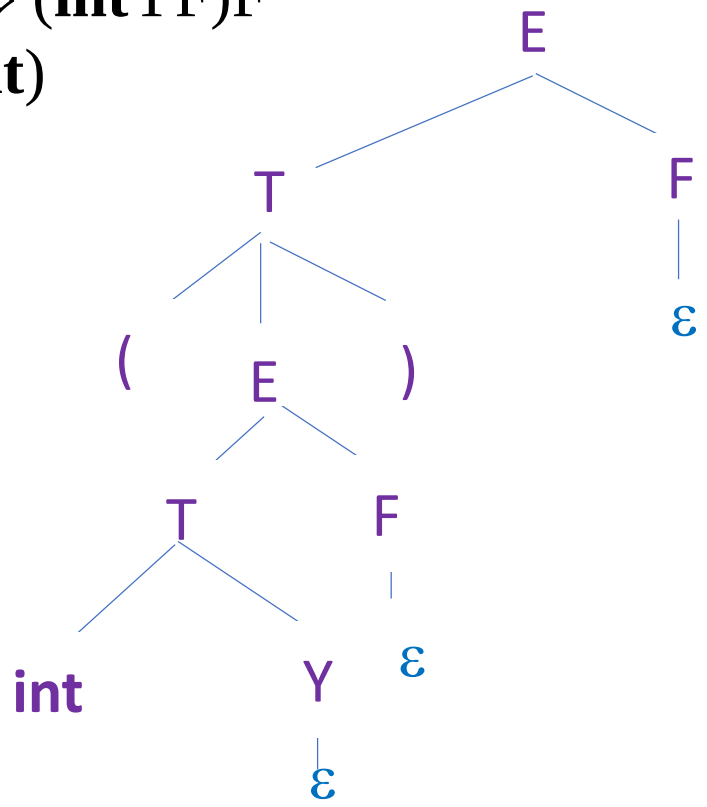
  $F \rightarrow +E \mid \varepsilon$

  $T \rightarrow \mathbf{int}Y \mid (E)$

  $Y \rightarrow *T \mid \varepsilon$

$$E \Rightarrow TF \Rightarrow (E)F \Rightarrow (TF)F \Rightarrow (\mathbf{int}YF)F \Rightarrow (\mathbf{int}F)F \Rightarrow (\mathbf{int})F \Rightarrow (\mathbf{int})$$

- Input: **(int)**
- Give a parse tree for the string



Example

- Consider the grammar
  E → TF
  F → +E | ε
  T → **int**Y | (E)
  Y → *T | ε
- Input: **int+int**

NON-TERMINAL	INPUT SYMBOL					
	+	**int**	(	)	*	$
E		TF	TF			
F	+E			ε		ε
T		**int**Y	(E)			
Y	ε			ε	*T	ε

MATCHED	STACK	INPUT	ACTION
	E$	**int+int**$	
	TF$	**int+int**$	Output E → TF
	intYF$	**int+int**$	Output T → **int**Y
int	YF$	**+int**$	Match **int**
int	F$	**+int**$	Output Y → ε
int	+E$	**+int**$	Output F → +E
int+	E$	**int**$	Match +
int+	TF$	**int**$	Output E → TF
int+	**int**YF$	**int**$	Output T → **int**Y
int+int	YF$	$	Match **int**
int+int	F$	$	Output Y → ε
int+int	$	$	Output F → ε

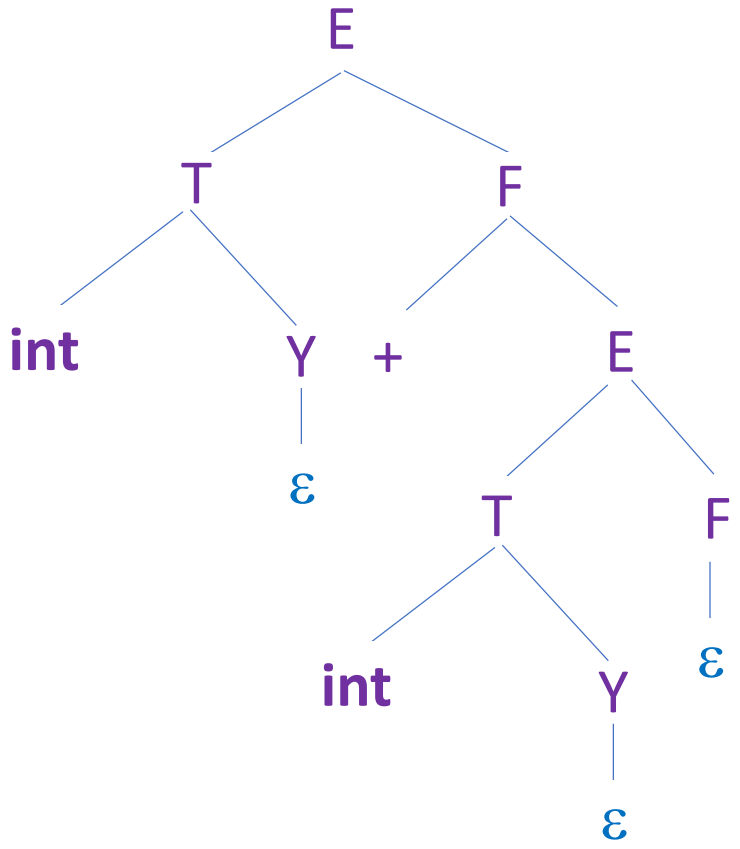
Example

- Consider the grammar

  $E \rightarrow TF$
  $F \rightarrow +E \mid \varepsilon$
  $T \rightarrow \mathbf{int}Y \mid (E)$
  $Y \rightarrow *T \mid \varepsilon$

$$E \Rightarrow TF \Rightarrow \mathbf{int}YF \Rightarrow \mathbf{int}F \Rightarrow \mathbf{int}+E \Rightarrow \mathbf{int}+TF \Rightarrow \mathbf{int}+\mathbf{int}YF \Rightarrow \mathbf{int}+\mathbf{int}F \Rightarrow \mathbf{int}+\mathbf{int}$$

- Input: **int+int**
- Give a parse tree for the string



LL(1) Parsing Table

- Consider the grammar: $S \rightarrow Sa \mid b$
- The parsing table

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do $M[A, \$] = \alpha$

	a	b	$
S		b Sa	

➢This grammar is not LL(1)

LL(1) Parsing Table

- If any entry is multiply defined then G is not LL(1)
  - Any grammar is not left factored, will not be LL(1)
  - Any grammar is left recursive, will not be LL(1)
  - Any grammar is ambiguous, will not be LL(1)
  - Grammar required more than one token of lookahead, will not be LL(1)
  - Other grammars are not LL(1) too

➢ Mechanical way to check that the grammar is LL(1)

- Build the LL(1) parsing table
- And see if all the entries in the table is unique

Exercise 1

- Consider the grammar

  $A \rightarrow AaE \mid E$

  $E \rightarrow E*T \mid b$

  $T \rightarrow aAb \mid c$

and the input string b*abb

- ➢ LL(1) grammar
- ➢ Show the parsing table
- ➢ Devise predictive parsers (for the input string)
- ➢ Give a parse tree (for the input string)

Exercise 2

- Consider the grammar

  $B \rightarrow FA$

  $F \rightarrow a\text{-}E \mid a\text{-}b \mid a\text{-}c$

  $E \rightarrow \text{-}Aa$

  $A \rightarrow b \mid c \mid \varepsilon$

and the input string a--cab

- LL(1) grammar
- Show the parsing table
- Devise predictive parsers (for the input string)
- Give a parse tree (for the input string)

Exercise 3

- Consider the grammar

  $S \rightarrow SE \mid Sb \mid a$

  $E \rightarrow +T \mid +b$

  $T \rightarrow (S) \mid c$

and the input string a+(ab)

- ➢ LL(1) grammar
- ➢ Show the parsing table
- ➢ Devise predictive parsers (for the input string)
- ➢ Give a parse tree (for the input string)

Exercise 1

- Consider the grammar

  $A \rightarrow AaE \mid E$

  $E \rightarrow E*T \mid b$

  $T \rightarrow aAb \mid c$

➢ LL(1) grammar

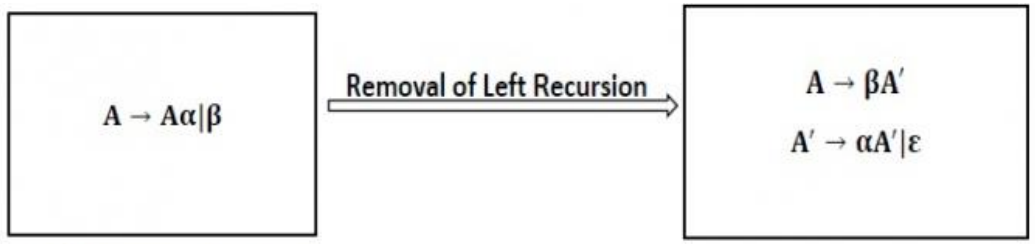
$A \rightarrow EA'$

$A' \rightarrow aEA' \mid \varepsilon$

$E \rightarrow bE'$

$E' \rightarrow *TE' \mid \varepsilon$

$T \rightarrow aAb \mid c$



Removal of Left Recursion

Exercise 1

- LL(1) grammar

$A \rightarrow EA'$

$A' \rightarrow aEA' \mid \varepsilon$

$E \rightarrow bE'$

$E' \rightarrow *TE' \mid \varepsilon$

$T \rightarrow aAb \mid c$

➢ FIRST, FOLLOW

Compute FIRST(X):
- If X is a terminal: FIRST(X) = {X}
- If X is a nonterminal:
  - $\varepsilon \in$ FIRST(X):
    - If $X \rightarrow \varepsilon$
    - If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all $1 \leq i \leq k$
  - $b \in$ FIRST(X):
    - If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \leq j \leq i-1$

FIRST (a) = {a}
FIRST (b) = {b}
FIRST (c) ={c}
FIRST (*) ={*}

FIRST(A)={b}
FIRST(A')={a, ε}
FIRST(E)={b}
FIRST(E')={*, ε}
FIRST(T)={a, c}

Compute FOLLOW(X):
- $ ∈ FOLLOW(S), where S is the start symbol, $ is the end symbol of the input string
- For each production $A \rightarrow \alpha X\beta$: FIRST(β)\{ε} ⊆ FOLLOW(X)
- For each production $A \rightarrow \alpha X\beta$, where ε∈FIRST(β): FOLLOW(A) ⊆ FOLLOW(X)
- For each production $A \rightarrow \alpha X$: FOLLOW(A) ⊆ FOLLOW(X)

FOLLOW(A)={$, b}
FOLLOW(A')={$, b}
FOLLOW(E)={a, $, b}
FOLLOW(E')={a, $, b}
FOLLOW(T)={*, a, $, b}

Exercise 1

- LL(1) grammar

$A \rightarrow EA'$

$A' \rightarrow aEA' \mid \varepsilon$

$E \rightarrow bE'$

$E' \rightarrow *TE' \mid \varepsilon$

$T \rightarrow aAb \mid c$

➢ Show the parsing table

FIRST (a) = {a}
FIRST (b) = {b}
FIRST (c) ={c}
FIRST (*) ={*}

FIRST(A)={b}
FIRST(A')={a, ε}
FIRST(E)={b}
FIRST(E')={*, ε}
FIRST(T)={a, c}

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do $M[A, \$] = \alpha$

NON-TERMINAL	INPUT SYMBOL				
	a	b	c	*	$
A		EA'			
A'	aEA'	ε			ε
E		bE'			
E'	ε	ε		*TE'	ε
T	aAb		c		

FOLLOW(A)={$, b}
FOLLOW(A')={$, b}
FOLLOW(E)={a, $, b}
FOLLOW(E')={a, $, b}
FOLLOW(T)={*, a, $, b}

FIRST(EA')=FIRST(E)={b}
FIRST(aEA')=FIRST(a) ={a}
FIRST(bE')=FIRST(b) ={b}
FIRST(*TE')= FIRST(*) ={*}
FIRST(aAb)=FIRST(a)={a}
FIRST(ε)= {ε}

Exercise 1

- LL(1) grammar

$A \rightarrow EA'$

$A' \rightarrow aEA' \mid \varepsilon$

$E \rightarrow bE'$

$E' \rightarrow *TE' \mid \varepsilon$

$T \rightarrow aAb \mid c$

➢ Devise predictive parsers for the input string **b*abb**

NON-TERMINAL	INPUT SYMBOL				
	a	b	c	*	$
A		EA'			
A'	aEA'	ε			ε
E		bE'			
E'	ε	ε		*TE'	ε
T	aAb		c		

MATCHED	STACK	INPUT	ACTION
	A$	**b*abb**$	
	EA'$	**b*abb**$	Output A → EA'
	bE'A'$	**b*abb**$	Output E → bE'
b	E'A'$	***abb**$	Match **b**
b	*TE'A'$	***abb**$	Output E' → *TE'
b*	TE'A'$	**abb**$	Match *
b*	aAbE'A'$	**abb**$	Output T → aAb
b*a	AbE'A'$	**bb**$	Match **a**
b*a	EA'bE'A'$	**bb**$	Output A → EA'
b*a	bE'A'bE'A'$	**bb**$	Output E → bE'
b*ab	E'A'bE'A'$	**b**$	Match **b**
b*ab	A'bE'A'$	**b**$	Output E' → ε
b*ab	bE'A'$	**b**$	Output A' → ε
b*abb	E'A'$	$	Match **b**
b*abb	A'$	$	Output E' → ε
b*abb	$	$	Output A' → ε

Exercise 1

- LL(1) grammar

$A \rightarrow EA'$

$A' \rightarrow aEA' \mid \varepsilon$

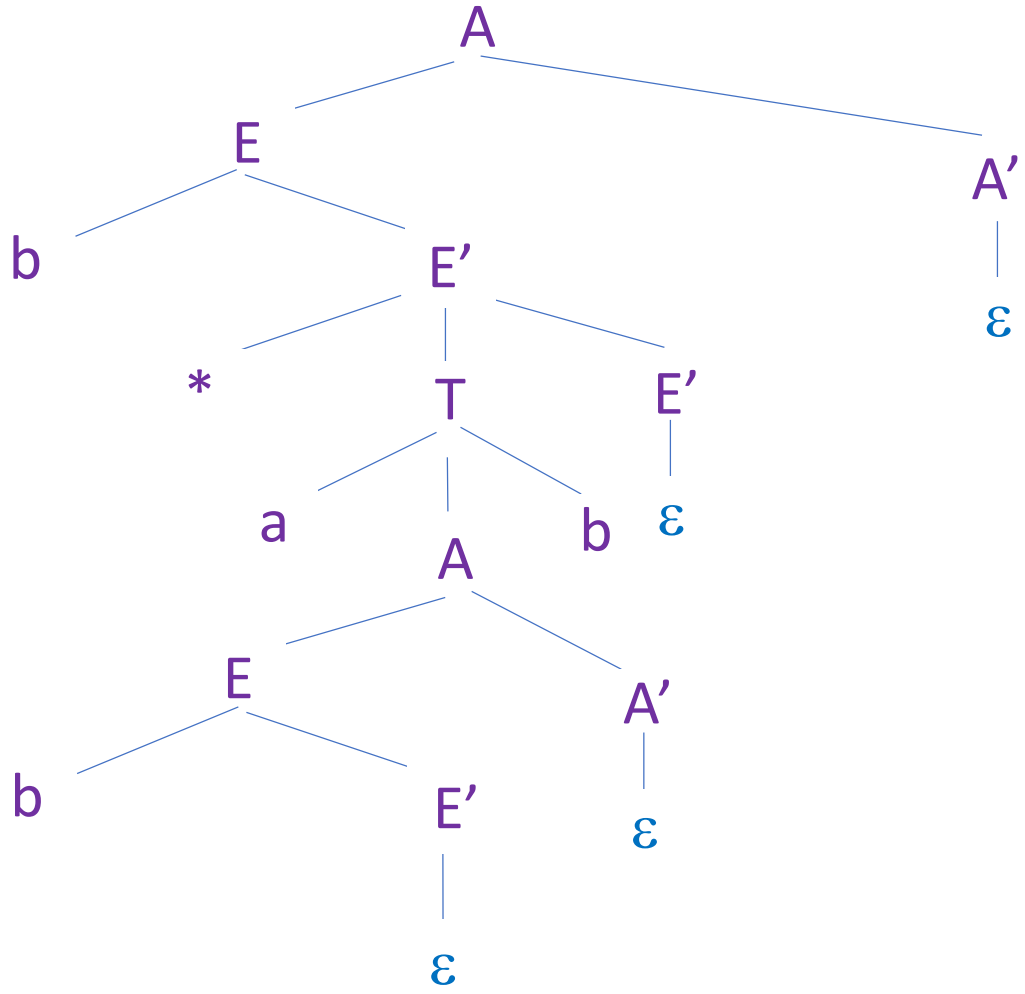
$E \rightarrow bE'$

$E' \rightarrow *TE' \mid \varepsilon$

$T \rightarrow aAb \mid c$

➢ Give a parse tree for the input string b*abb

$A \Rightarrow EA' \Rightarrow bE'A' \Rightarrow b*TE'A' \Rightarrow b*aAbE'A'$
$\Rightarrow b*aEA'bE'A' \Rightarrow b*abE'A'bE'A'$
$\Rightarrow b*abA'bE'A' \Rightarrow b*abbE'A' \Rightarrow b*abbA'$
$\Rightarrow b*abb$



Exercise 2

- Consider the grammar

  $B \rightarrow FA$

  $F \rightarrow a\text{-}E \mid a\text{-}b \mid a\text{-}c$

  $E \rightarrow -Aa$

  $A \rightarrow b \mid c \mid \varepsilon$

➢ LL(1) grammar

$B \rightarrow FA$

$F \rightarrow a\text{-}F'$

$F' \rightarrow E \mid b \mid c$

$E \rightarrow -Aa$

$A \rightarrow b \mid c \mid \varepsilon$

Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$
and the input begins with a nonempty string derived from α

- ➢ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
- ➢ we may defer the decision by
  - expanding A to $\alpha A'$
  - then exapanding A' to β_1 or β_2
- ➢ the original productions become

  $A \rightarrow \alpha A'$

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

Exercise 2

- LL(1) grammar

$B \rightarrow FA$

$F \rightarrow a\text{-}F'$

$F' \rightarrow E \mid b \mid c$

$E \rightarrow \text{-}Aa$

$A \rightarrow b \mid c \mid \varepsilon$

➢ FIRST, FOLLOW

Compute FIRST(X):
- If X is a terminal: FIRST(X) = {X}
- If X is a nonterminal:
  - $\varepsilon \in$ FIRST(X):
    - If $X \rightarrow \varepsilon$
    - If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all $1 \le i \le k$
  - $b \in$ FIRST(X):
    - If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \le j \le i\text{-}1$

Compute FOLLOW(X):
- $ ∈ FOLLOW(S), where S is the start symbol, $ is the end symbol of the input string
- For each production $A \rightarrow \alpha X\beta$: FIRST(β)\{ε} ⊆ FOLLOW(X)
- For each production $A \rightarrow \alpha X\beta$, where ε∈FIRST(β): FOLLOW(A) ⊆ FOLLOW(X)
- For each production $A \rightarrow \alpha X$: FOLLOW(A) ⊆ FOLLOW(X)

FIRST (a) = {a}
FIRST (b) = {b}
FIRST (c) ={c}
FIRST (-) ={-}

FIRST(B)={a }
FIRST(A)={b, c, ε}
FIRST(F)={a}
FIRST(F')={-, b, c}
FIRST(E)={-}

FOLLOW(B)={$}
FOLLOW(A)={a, $}
FOLLOW(F)={b, c, $}
FOLLOW(F')={b, c, $}
FOLLOW(E)={b, c, $}

Exercise 2

- LL(1) grammar

$B \rightarrow FA$

$F \rightarrow a\text{-}F'$

$F' \rightarrow E \mid b \mid c$

$E \rightarrow -Aa$

$A \rightarrow b \mid c \mid \varepsilon$

➢ Show the parsing table

FIRST (a) = {a}
FIRST (b) = {b}
FIRST (c) ={c}
FIRST (-) ={-}

FIRST(B)={a }
FIRST(A)={b, c, ε}
FIRST(F)={a}
FIRST(F')={-, b, c}
FIRST(E)={-}

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do $M[A, \$] = \alpha$

NON-TERMINAL	INPUT SYMBOL				
	a	b	c	-	$
B	FA				
A	ε	b	c		ε
F	a-F'				
F'		b	c	E	
E				-Aa	

FOLLOW(B)={$}
FOLLOW(A)={a, $}
FOLLOW(F)={b, c, $}
FOLLOW(F')={b, c, $}
FOLLOW(E)={b, c, $}

FIRST(FA)=FIRST(F)={a}
FIRST(a-F')=FIRST(a) ={a}
FIRST(-Aa)=FIRST(-) ={-}
FIRST(ε)= {ε}

Exercise 2

- LL(1) grammar

B → FA

F → a-F’

F’ → E | b | c

E → -Aa

A → b | c | ε

➢ Devise predictive parsers for the input string **a--cab**

NON-TERMINAL	INPUT SYMBOL				
	a	b	c	-	$
B	FA				
A	ε	b	c		ε
F	a-F’				
F’		b	c	E	
E				-Aa	

MATCHED	STACK	INPUT	ACTION
	B$	**a--cab**$	
	FA$	**a--cab**$	Output B → FA
	a-F’A$	**a--cab**$	Output F → a-F’
a	-F’A$	**--cab**$	Match **a**
a-	F’A$	**-cab**$	Match -
a-	EA$	**-cab**$	Output F’ → E
a-	-AaA$	**-cab**$	Output E → -Aa
a--	AaA$	**cab**$	Match -
a--	caA$	**cab**$	Output A → c
a--c	aA$	**ab**$	Match **c**
a--ca	A$	**b**$	Match **a**
a--ca	b$	**b**$	Output A → b
a--cab	$	$	Match **b**

Exercise 2

- LL(1) grammar

$B \rightarrow FA$

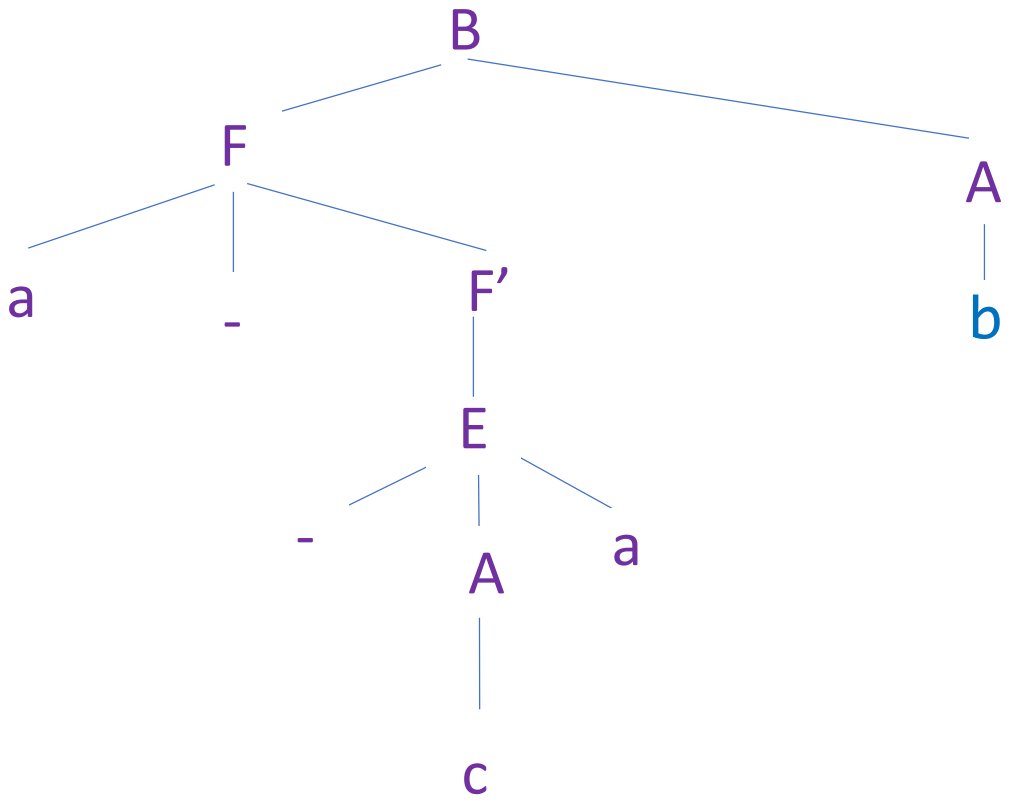
$F \rightarrow a\text{-}F'$

$F' \rightarrow E \mid b \mid c$

$E \rightarrow \text{-}Aa$

$A \rightarrow b \mid c \mid \varepsilon$

➢ Give a parse tree for the input string a--cab



$B \Rightarrow FA \Rightarrow a\text{-}F'A \Rightarrow a\text{-}F'A \Rightarrow a\text{-}EA$

$\Rightarrow a\text{--}AaA \Rightarrow a\text{--}caA \Rightarrow a\text{--}cab$

Exercise 3

- Consider the grammar

  $S \rightarrow SE \mid Sb \mid a$

  $E \rightarrow +T \mid +b$

  $T \rightarrow (S) \mid c$

➢ LL(1) grammar

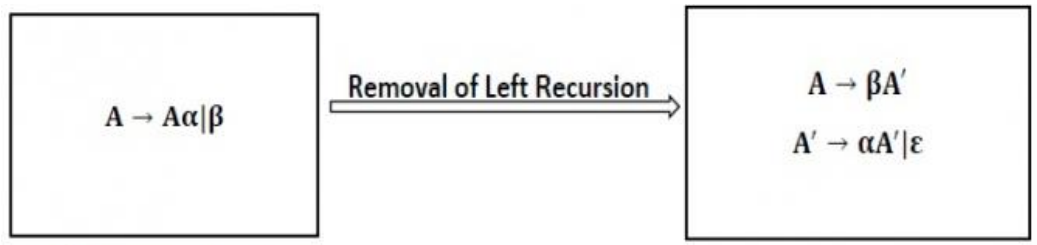
$S \rightarrow aS'$

$S' \rightarrow ES' \mid bS' \mid \varepsilon$

$E \rightarrow +E'$

$E' \rightarrow T \mid b$

$T \rightarrow (S) \mid c$



Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$
and the input begins with a nonempty string derived from α

- ➢ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
- ➢ we may defer the decision by
  - expanding A to $\alpha A'$
  - then exapanding A' to β_1 or β_2
- ➢ the original productions become

  $A \rightarrow \alpha A'$

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

Exercise 3

- LL(1) grammar

$S \rightarrow aS'$

$S' \rightarrow ES' \mid bS' \mid \varepsilon$

$E \rightarrow +E'$

$E' \rightarrow T \mid b$

$T \rightarrow (S) \mid c$

➢ FIRST, FOLLOW

Compute FIRST(X):
- If X is a terminal: FIRST(X) = {X}
- If X is a nonterminal:
  - $\varepsilon \in$ FIRST(X):
    - If $X \rightarrow \varepsilon$
    - If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all $1 \leq i \leq k$
  - $b \in$ FIRST(X):
    - If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \leq j \leq i-1$

FIRST (a) = {a}
FIRST (b) = {b}
FIRST (c) = {c}
FIRST (+) ={+}
FIRST (() ={(}
FIRST ()) ={)}

FIRST(S)={a}
FIRST(S')={+, b, ε}
FIRST(E)={+}
FIRST(E')={(, c, b}
FIRST(T)={(, c}

Compute FOLLOW(X):
- $ ∈ FOLLOW(S), where S is the start symbol, $ is the end symbol of the input string
- For each production $A \rightarrow \alpha X\beta$: FIRST(β)\{ε} ⊆ FOLLOW(X)
- For each production $A \rightarrow \alpha X\beta$, where ε∈FIRST(β): FOLLOW(A) ⊆ FOLLOW(X)
- For each production $A \rightarrow \alpha X$: FOLLOW(A) ⊆ FOLLOW(X)

FOLLOW(S)={$,)}
FOLLOW(S')={$,)}
FOLLOW(E)={+, b, $,)}
FOLLOW(E')={+, b, $,)}
FOLLOW(T)={+, b, $,)}

Exercise 3

- LL(1) grammar

$S \rightarrow aS'$

$S' \rightarrow ES' \mid bS' \mid \varepsilon$

$E \rightarrow +E'$

$E' \rightarrow T \mid b$

$T \rightarrow (S) \mid c$

➢ Show the parsing table

FIRST (a) = {a}
FIRST (b) = {b}
FIRST (c) = {c}
FIRST (+) ={+}
FIRST (() ={(}
FIRST ()) ={)}

FIRST(S)={a}
FIRST(S')={+, b, ε}
FIRST(E)={+}
FIRST(E')={(, c, b}
FIRST(T)={(, c}

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do $M[A, \$] = \alpha$

NON-TERMINAL	INPUT SYMBOL						
	a	b	c	+	(	)	$
S	aS'						
S'		bS'		ES'		ε	ε
E				+E'			
E'		b	T		T		
T			c		(S)		

FOLLOW(S)={$,)}
FOLLOW(S')={$,)}
FOLLOW(E)={+, b, $,)}
FOLLOW(E')={+, b, $,)}
FOLLOW(T)={+, b, $,)}

FIRST(aS')=FIRST(a)={a}
FIRST(ES')=FIRST(E) ={+}
FIRST(bS')=FIRST(b) ={b}
FIRST(+E')= FIRST(+) ={+}
FIRST((S))=FIRST(()={(}
FIRST(ε)= {ε}

Exercise 3

- LL(1) grammar

$S \rightarrow aS'$

$S' \rightarrow ES' \mid bS' \mid \varepsilon$

$E \rightarrow +E'$

$E' \rightarrow T \mid b$

$T \rightarrow (S) \mid c$

➢ Devise predictive parsers for the input string **a+(ab)**

NON-TERMINAL	INPUT SYMBOL						
	a	b	c	+	(	)	$
S	aS'						
S'		bS'		ES'		ε	ε
E				+E'			
E'		b	T		T		
T			c		(S)		

MATCHED	STACK	INPUT	ACTION
	S$	**a+(ab)**$	
	aS'$	**a+(ab)**$	Output S → aS'
a	S'$	**+(ab)**$	Match **a**
a	ES'$	**+(ab)**$	Output S' → ES'
a	+E'S'$	**+(ab)**$	Output E → +E'
a+	E'S'$	**(ab)**$	Match +
a+	TS'$	**(ab)**$	Output E' → T
a+	(S)S'$	**(ab)**$	Output T → (S)
a+(	S)S'$	**ab)**$	Match (
a+(	aS')S'$	**ab)**$	Output S → aS'
a+(a	S')S'$	**b)**$	Match **a**
a+(a	bS')S'$	**b)**$	Output S' → bS'
a+(ab	S')S'$	**)**$	Match **b**
a+(ab	)S'$	**)**$	Output S' → ε
a+(ab)	S'$	$	Match **)**
a+(ab)	$	$	Output S' → ε

Exercise 3

- LL(1) grammar

$S \rightarrow aS'$

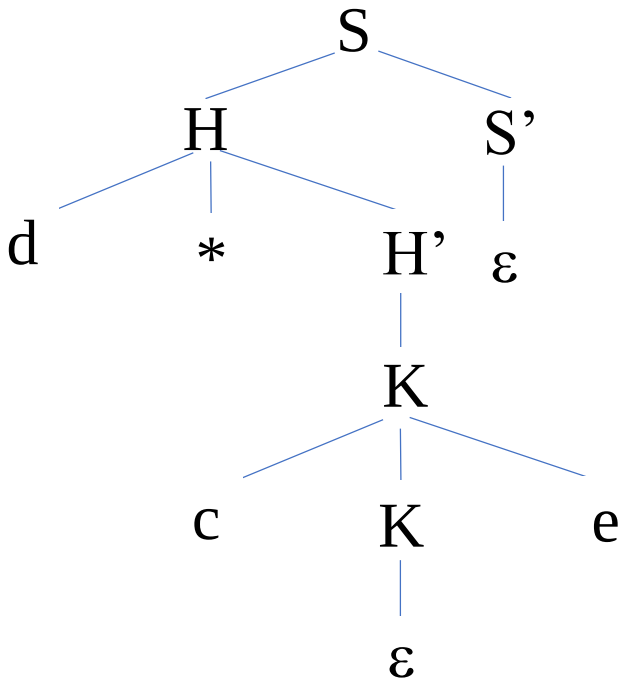
$S' \rightarrow ES' \mid bS' \mid \varepsilon$

$E \rightarrow +E'$

$E' \rightarrow T \mid b$

$T \rightarrow (S) \mid c$

$S \Rightarrow aS' \Rightarrow aES' \Rightarrow a+E'S'$
$\Rightarrow a+TS' \Rightarrow a+(S)S'$
$\Rightarrow a+(aS')S' \Rightarrow a+(abS')S'$
$\Rightarrow a+(ab)S' \Rightarrow a+(ab)$



➢ Give a parse tree for the input string a+(ab)