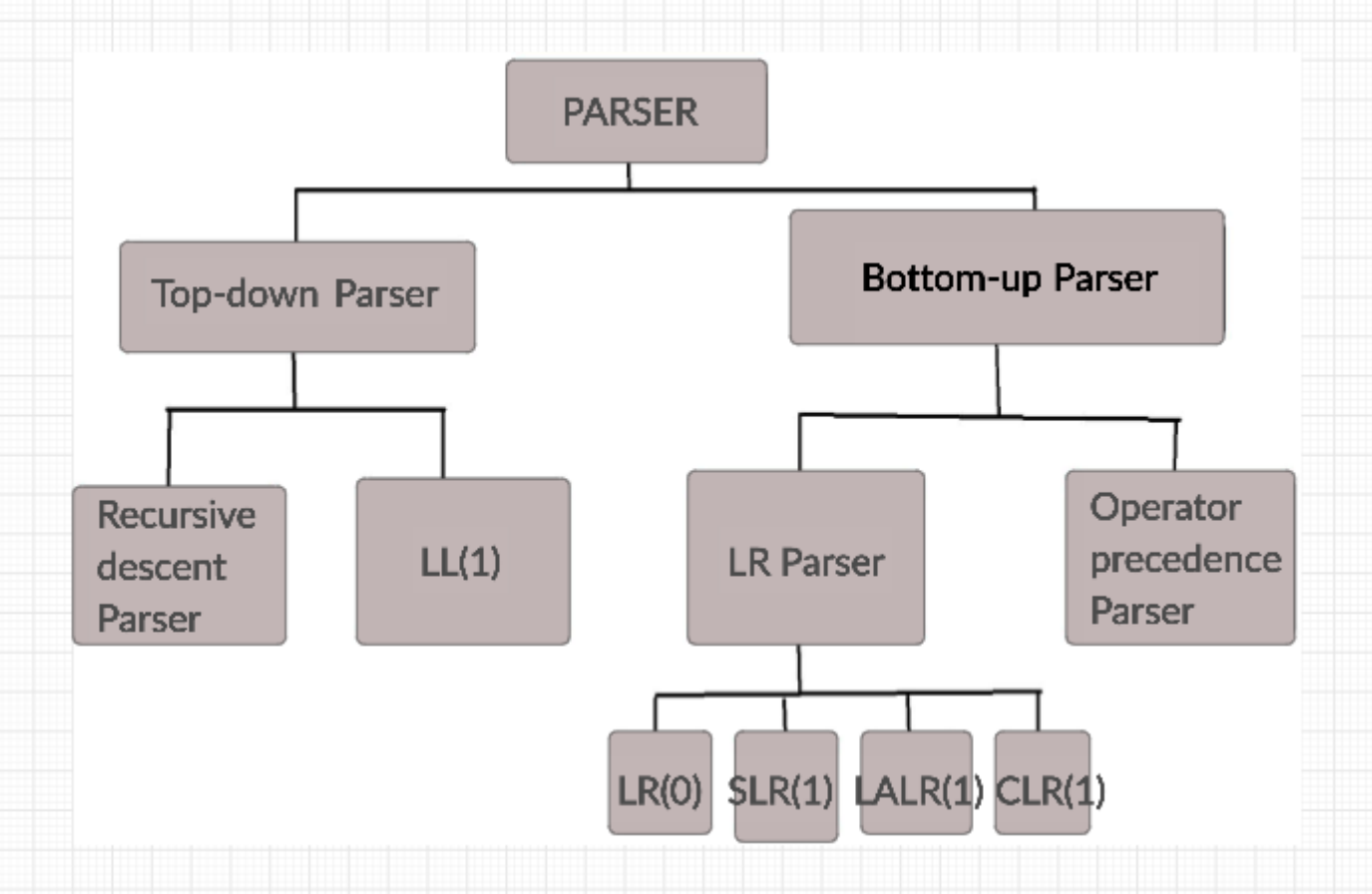
Compiler Construction

SYNTAX ANALYSIS

TYPES OF PARSER



PREDICTIVE PARSER

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$

input: … a …….

current token

- Unlike recursive-descent, predictive parser can "predict" which production to use.
  - By looking at the next few tokens.
  - No backtracking.

PREDICTIVE PARSER (EXAMPLE)

stmt → `if` ..... |
`while` ..... |
`begin` ..... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.

Predictive Parsing

Will never backtrack!

Requirement:

For every rule:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_N$$

We must be able to choose the correct alternative by looking only at the next symbol

May peek ahead to the next symbol (token).

Example

A → **a**B
→ **c**D
→ E

Assuming **a**,**c** ∉ FIRST (E)

Example

Stmt → **if** Expr ...
→ **for** LValue ...
→ **while** Expr ...
→ **return** Expr ...
→ ID ...

Predictive Parsing

LL(1) Grammars

Can do predictive parsing
Can select the right rule
Looking at only the next 1 input symbol

LL(k) Grammars

Can do predictive parsing
Can select the right rule
Looking at only the next k input symbols

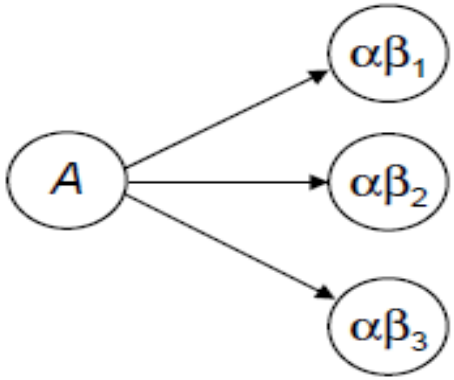
Techniques to modify the grammar:

- Left Factoring
- Removal of Left Recursion

Left Factoring

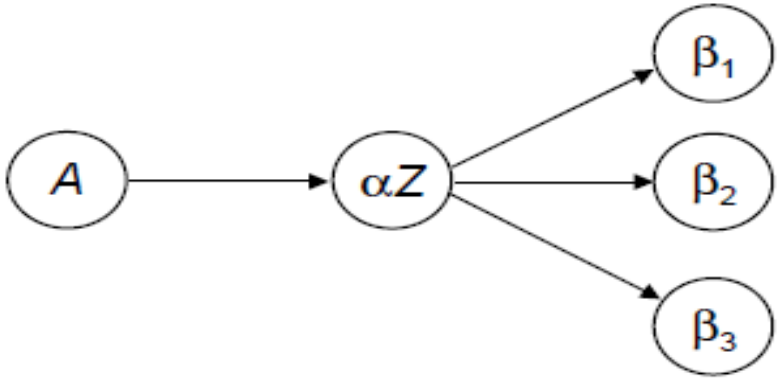
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1$
$\quad | \ \alpha\beta_2$
$\quad | \ \alpha\beta3$



becomes ...

$A \rightarrow \alpha\ Z$
$Z \rightarrow \beta_1$
$\quad | \ \beta_2$
$\quad | \ \beta_n$



Left Factoring (An example)

Consider the following fragment of the expression grammar

Factor	→	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

After left factoring, it becomes

Factor	→	Identifier*Arguments*
Argumen	→	[*ExprList*]
	|	(*ExprList*)
	|	ε

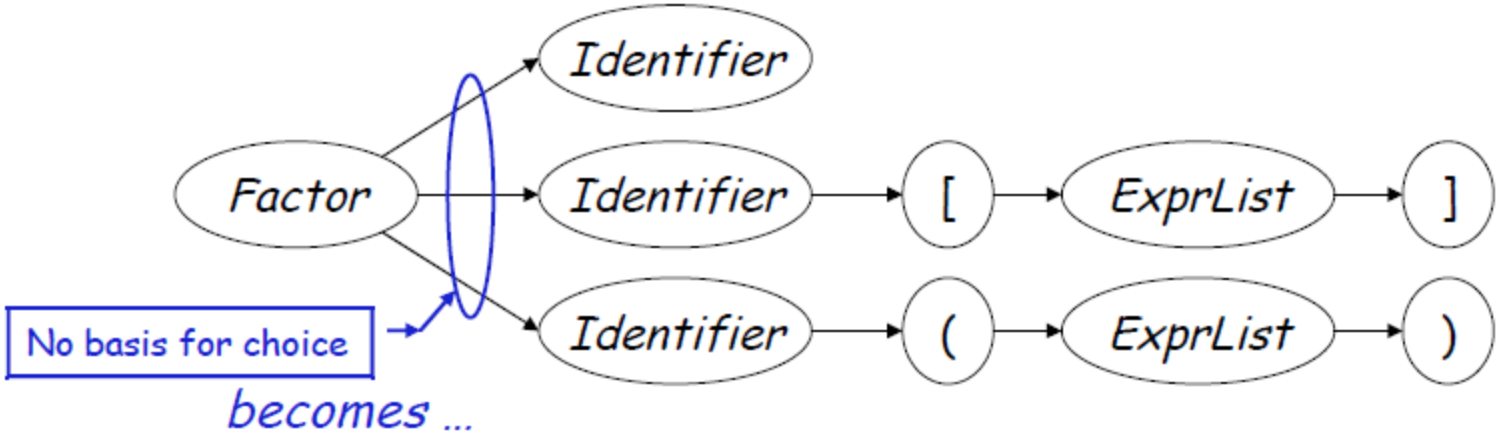
FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)

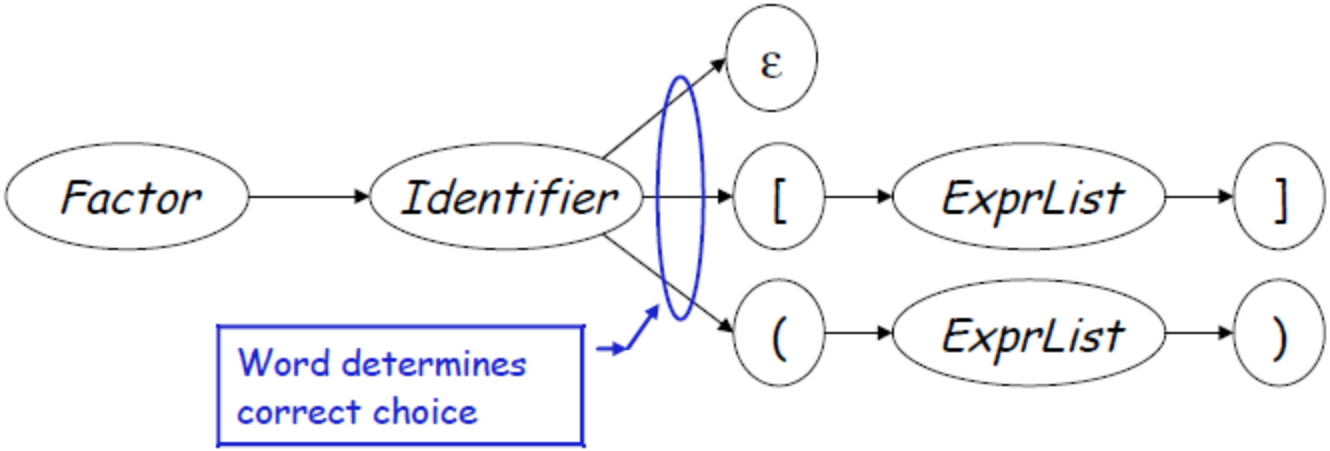
⇒ It has the *LL(1)* property

This form has the same syntax, with the *LL(1)* property

Left Factoring

Graphically





Left-Factoring

Problem:

Stmt → **`if`** Expr **`then`** Stmt **`else`** Stmt

→ **`if`** Expr **`then`** Stmt

→ OtherStmt

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → **`if`** Expr **`then`** Stmt ElsePart

→ OtherStmt

ElsePart → **`else`** Stmt | ε

General Approach:

Before: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: $A \rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

$C \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left-Factoring

Problem:

Stmt → **if** Expr **then** Stmt **else** Stmt

Stmt → **if** Expr **then** Stmt ε

Stmt → OtherStmt

(A = Stmt; α = **if** Expr **then** Stmt; β1 = **else** Stmt; β2 = ε; δ1 = OtherStmt)

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → **if** Expr **then** Stmt ElsePart

Stmt → OtherStmt

ElsePart → **else** Stmt | ε

(A = Stmt; α = **if** Expr **then** Stmt; C = ElsePart; δ1 = OtherStmt; β1 = **else** Stmt; β2 = ε)

General Approach:

Before: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: $A \rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

$C \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left Recursion

Whenever

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

Simplest Case: Immediate Left Recursion

Given:

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

Transform into:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$ where A' is a new nonterminal

More General (but still immediate):

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Transform into:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. S ⇒ Af ⇒ Sdf

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

A → Sd

Get rid of the S. Substitute in the righthand sides of S.

A → Afd | bd

The modified grammar:

S → Af | b

A → Ac | Afd | bd | e

Now eliminate immediate left recursion involving A.

S → Af | b

A → bdA' | eA'

A' → cA' | fdA' | ε

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**

A → A**c** | S**d** | B**e**

B → A**g** | S**h** | **k**

Assume there are still more nonterminals; Look at the next one...

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**

A → A**c** | S**d** | B**e**

B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**

A → **bd**A' | B**e**A'

A' → **c**A' | **fd**A' | **ε**

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**

A → A**c** | S**d** | B**e**

B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**

A → **bd**A' | B**e**A'

A' → **c**A' | **fd**A' | **ε**

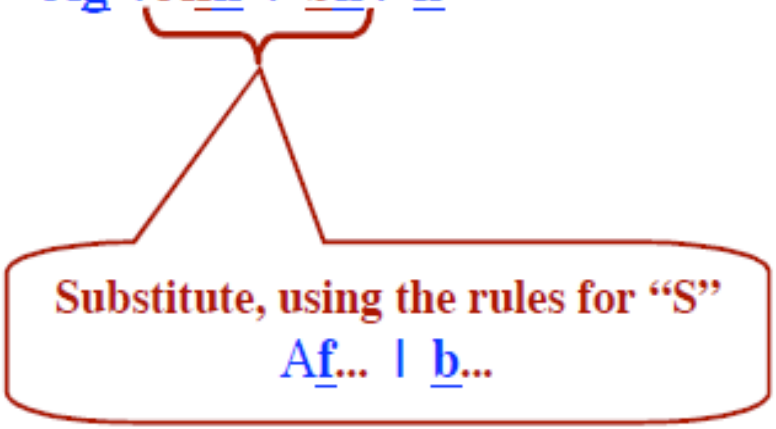
Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Afh | bh | k



Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**

A → A**c** | S**d** | B**e**

B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**

A → **bd**A' | B**e**A'

A' → **c**A' | **fd**A' | ε

B → A**g** | A**fh** | **bh** | **k**

Does any righthand side start with "A"?

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**

A → A**c** | S**d** | B**e**

B → A**g** | S**h** | **k**

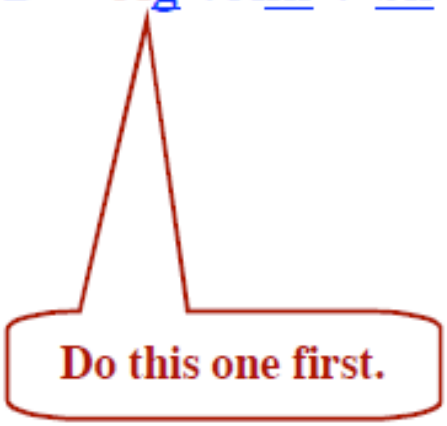
So Far:

S → A**f** | **b**

A → **bd**A' | B**e**A'

A' → **c**A' | **fd**A' | ε

B → A**g** | A**fh** | **bh** | **k**



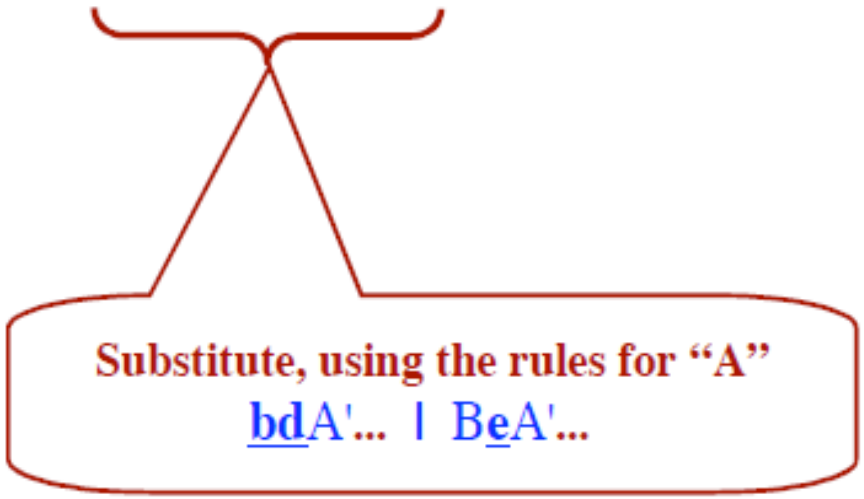
Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | Afh | bh | k



Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**

A → A**c** | S**d** | B**e**

B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**

A → **bd**A' | B**e**A'

A' → **c**A' | **fd**A' | ε

B → **bd**A'**g** | B**e**A'**g** | A**fh** | **bh** | **k**

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b

A → Ac | Sd | Be

B → Ag | Sh | k

So Far:

S → Af | b

A → bdA' | BeA'

A' → cA' | fdA' | ε

B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Substitute, using the rules for "A"

bdA'... | BeA'...

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b

A → Ac | Sd | Be

B → Ag | Sh | k

So Far:

S → Af | b

A → bdA' | BeA'

A' → cA' | fdA' | ε

B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Finally, eliminate any immediate Left recursion involving "B"

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b

A → Ac | Sd | Be

B → Ag | Sh | k

So Far:

S → Af | b

A → bdA' | BeA'

A' → cA' | fdA' | ε

B → bdA'gB' | bdA'fhB' | bhB' | kB'

B' → eA'gB' | eA'fhB' | ε

Finally, eliminate any immediate Left recursion involving "B"

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b

A → Ac | Sd | Be | C

B → Ag | Sh | k

C → BkmA | AS | j

If there is another nonterminal, then do it next.

So Far:

S → Af | b

A → bdA' | BeA' | CA'

A' → cA' | fdA' | ε

B → bdA'gB' | bdA'fhB' | bhB' | kB' | CA'gB' | CA'fhB'

B' → eA'gB' | eA'fhB' | ε

Algorithm to Eliminate Left Recursion

Assume the nonterminals are ordered A_1, A_2, A_3, ...

(In the example: S, A, B)

for each nonterminal A_i (for i = 1 to N) do

  for each nonterminal A_j (for j = 1 to i-1) do

    Let $A_j \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_N$ be all the rules for A_j

    if there is a rule of the form

      $A_i \rightarrow A_j\alpha$

    then replace it by

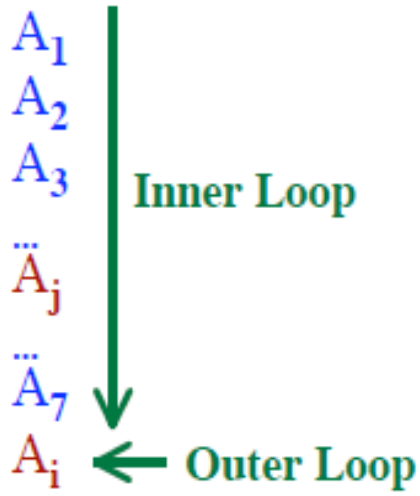
      $A_i \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid \beta_3\alpha \mid \ldots \mid \beta_N\alpha$

    endIf

  endFor

  Eliminate immediate left recursion among the A_i rules

endFor



CONSTRUCTING THE LL(1) PARSING TABLE

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

$= \{ \mathtt{a} \mid \alpha \Rightarrow^* \mathtt{aw}$, plus ε if $\alpha \Rightarrow^* \varepsilon \}$

To Compute the "FIRST" Function

For all symbols X in the grammar...

```
if X is a terminal then
  FIRST(X) = { X }

if X → ε is a rule then
  add ε to FIRST(X)

if X → Y1 Y2 Y3 ... YK is a rule then
  if a ∈ FIRST(Y1) then
    add a to FIRST(X)
  if ε ∈ FIRST(Y1) and a ∈ FIRST(Y2) then
    add a to FIRST(X)
  if ε ∈ FIRST(Y1) and ε ∈ FIRST(Y2) and a ∈ FIRST(Y3) then
    add a to FIRST(X)
  ...
  if ε ∈ FIRST(Yi) for all Yi then
    add ε to FIRST(X)
```

Repeat until nothing more can be added to any sets.

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
```

```
endIf
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
  endIf
endIf
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result
    endIf
  endIf
endIf
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result
      ...
        if ε ∈ FIRST(XN-1) then
          Add everything in FIRST(XN), except ε, to result




        endIf
      ...
    endIf
  endIf
endIf
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result
      ...
        if ε ∈ FIRST(XN-1) then
          Add everything in FIRST(XN), except ε, to result
          if ε ∈ FIRST(XN) then
            // Then X1 ⇒* ε, X2 ⇒* ε, X3 ⇒* ε, ... XN ⇒* ε
            Add ε to result
          endIf
        endIf
      ...
    endIf
  endIf
endIf
```

EXAMPLE

$A \rightarrow BC$

$B \rightarrow DE$

$D \rightarrow FG$

$F \rightarrow HI$

$H \rightarrow xY$

First(A) = {x}

Example for computing FIRST(α)

Grammar

$E \rightarrow E'T$

$E' \rightarrow -TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow /FT' \mid \epsilon$

$F \rightarrow int \mid (E)$

$\boldsymbol{FIRST}(F) = \{int, (\}$

$\boldsymbol{FIRST}(T') = \{/, \epsilon\}$

$\boldsymbol{FIRST}(T) = \{int, (\}$, since $\epsilon \notin \boldsymbol{FIRST}(F)$, that's all.

$\boldsymbol{FIRST}(E') = \{-, \epsilon\}$

$\boldsymbol{FIRST}(E) = \{-, int, (\}$, since $\epsilon \in \boldsymbol{FIRST}(E')$.

$\boldsymbol{FIRST}(\epsilon) = \{\epsilon\}$

$\boldsymbol{FIRST}(E'T) = \{-, int, (\}$

$\boldsymbol{FIRST}(-TE') = \{-\}$

$\boldsymbol{FIRST}(\epsilon) = \{\epsilon\}$

$\boldsymbol{FIRST}(FT') = \{int,)\}$

$\boldsymbol{FIRST}(/FT') = \{/\}$

$\boldsymbol{FIRST}(\epsilon) = \{\epsilon\}$

$\boldsymbol{FIRST}(int) = \{int\}$

$\boldsymbol{FIRST}((E)) = \{(\}$

- **FIRST**$(T'E') =$
  - $(\boldsymbol{FIRST}(T') - \{\epsilon\}) \cup$
  - $(\boldsymbol{FIRST}(E') - \{\epsilon\}) \cup$
  - $\{\epsilon\}$

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

= { **a** | $\alpha \Rightarrow^*$ **a**w, plus **ε** if $\alpha \Rightarrow^*$ **ε** }

Example:

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

$F \rightarrow (\ E\ ) \mid \underline{\texttt{id}}$

FIRST (F) = ?

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

= { **a** | $\alpha \Rightarrow^*$ **aw**, plus **ε** if $\alpha \Rightarrow^*$ **ε** }

Example:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

FIRST (F) = { (, **id** }

FIRST (T') = ?

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

= { **a** | $\alpha \Rightarrow^*$ **aw**, plus **ε** if $\alpha \Rightarrow^*$ **ε** }

Example:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

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

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

= { **a** | $\alpha \Rightarrow^*$ **a**w, plus **ε** if $\alpha \Rightarrow^*$ **ε** }

Example:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

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

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

= { **a** | $\alpha \Rightarrow^*$ **a**w, plus **ε** if $\alpha \Rightarrow^*$ **ε** }

Example:

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

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

FIRST (F) = { (, **id** }

FIRST (T') = { *, **ε**}

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

FIRST (E') = { +, **ε**}

FIRST (E) = ?

"FIRST" Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

= { **a** | $\alpha \Rightarrow^*$ **a**w, plus **ε** if $\alpha \Rightarrow^*$ **ε** }

Example:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

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

Why do we need FIRST(α)?

- During parsing, suppose top-of-stack is a nonterminal A and there are several choices
  - $A \rightarrow \alpha_1$
  - $A \rightarrow \alpha_2$
  - $\cdots$
  - $A \rightarrow \alpha_k$

  for derivation, and the current lookahead token is a
- If $a \in$ FIRST(α_i), then pick $A \rightarrow \alpha_i$ for derivation, pop, and then push α_i.
- If a is in several FIRST(α_i)'s, then the grammar is not $LL(1)$.
- Question: if a is not in any FIRST(α_i), does this mean the input stream cannot be accepted?
  - Maybe not!
  - What happen if ϵ is in some FIRST(α_i)?

TASK

Write the sets of the following:

S -> Ty

T -> AB

T -> sT

A -> aA

A -> λ

B -> bB

B -> λ

To Compute FOLLOW(A_i) for all Nonterminals in the Grammar

```
add $ to FOLLOW(S)
repeat
  if A→αBβ is a rule then
    add every terminal in FIRST(β) except ε to FOLLOW(B)
    if FIRST(β) contains ε then
      add everything in FOLLOW(A) to FOLLOW(B)
    endIf
  endIf
  if A→αB is a rule then
    add everything in FOLLOW(A) to FOLLOW(B)
  endIf
until We cannot add anything more
```

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \mathtt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { ?
FOLLOW (E') = { ?
FOLLOW (T) = { ?
FOLLOW (T') = { ?
FOLLOW (F) = { ?

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = {
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

Add $ to FOLLOW(S)

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { $,
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

Add $ to FOLLOW(S)

Example of FOLLOW Computation

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { $,
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule

F → (E) | id

What can follow E?

Example of FOLLOW Computation

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule

F → (E) | id

What can follow E?

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

Look at rule

$E \rightarrow T\ E'$

Whatever can follow E can also follow E'

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \mathtt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

Look at rule

$E \rightarrow T\ E'$

Whatever can follow E can also follow E'

Example of FOLLOW Computation

Previously computed FIRST sets...

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

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule

$E'_0 \rightarrow + T E'_1$

Whatever is in FIRST(E'_1) can follow T

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Look at rule

$E'_0 \rightarrow +\ T\ E'_1$

Whatever is in FIRST(E'_1) can follow T

Example of FOLLOW Computation

Previously computed FIRST sets...

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

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

The FOLLOW sets...

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

Look at rule

$T'_0 \rightarrow * F T'_1$

Whatever is in FIRST(T'_1) can follow F

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \mathtt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Look at rule

$T'_0 \rightarrow *\ F\ T'_1$

Whatever is in FIRST(T'_1) can follow F

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Look at rule

$E'_0 \rightarrow +\ T\ E'_1$

Since E'_1 can go to ε

i.e., $\varepsilon \in$ FIRST(E')

Everything in FOLLOW(E'_0) can follow T

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Look at rule

$E'_0 \rightarrow +\ T\ E'_1$

Since E'_1 can go to ε

i.e., $\varepsilon \in$ FIRST(E')

Everything in FOLLOW(E'_0) can follow T

Example of FOLLOW Computation

Previously computed FIRST sets...

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

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

The FOLLOW sets...

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

Look at rule

T → F T'

Whatever can follow T can also follow T'

Example of FOLLOW Computation

Previously computed FIRST sets...

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

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

The FOLLOW sets...

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

Look at rule

T → F T'

Whatever can follow T can also follow T'

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Look at rule

$T'_0 \rightarrow *\ F\ T'_1$

Since T'_1 can go to ε

i.e., $\varepsilon \in$ FIRST(T')

Everything in FOLLOW(T'_0) can follow F

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \texttt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Look at rule

$T'_0 \rightarrow *\ F\ T'_1$

Since T'_1 can go to ε

i.e., $\varepsilon \in$ FIRST(T')

Everything in FOLLOW(T'_0) can follow F

Example of FOLLOW Computation

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \mathtt{id}
\end{aligned}
$$

Previously computed FIRST sets...

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

The FOLLOW sets...

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

Nothing more can be added.

A complete example

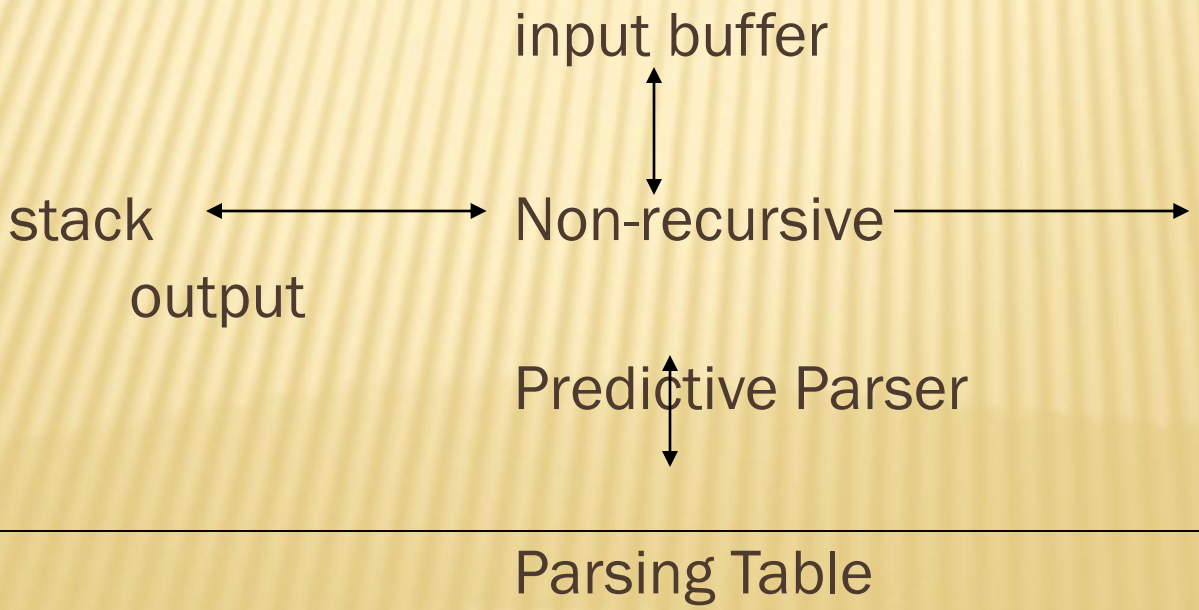
- **Grammar**
  - $S \rightarrow Bc \mid DB$
  - $B \rightarrow ab \mid cS$
  - $D \rightarrow d \mid \epsilon$

α	**FIRST**(α)	**FOLLOW**(α)
D	$\{d, \epsilon\}$	$\{a, c\}$
B	$\{a, c\}$	$\{c, \$\}$
S	$\{a, c, d\}$	$\{c, \$\}$
Bc	$\{a, c\}$	
DB	$\{d, a, c\}$	
ab	$\{a\}$	
cS	$\{c\}$	
d	$\{d\}$	
ϵ	$\{\epsilon\}$	

NON-RECURSIVE PREDICTIVE PARSING – LL(1) PARSER

NON-RECURSIVE PREDICTIVE PARSING – LL(1) PARSER

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



EXAMPLE PARSE TABLE CONSTRUCTION

$S \rightarrow B\ c \mid D\ B$

$B \rightarrow a\ b \mid c\ S$

$D \rightarrow d \mid \varepsilon$

For this grammar:

- Construct FIRST and FOLLOW Sets
- Apply algorithm to calculate parse table

EXAMPLE PARSE TABLE CONSTRUCTION

X	FIRST(X)	FOLLOW(X)
D	{ d, **ε** }	{ a, c }
B	{ a, c }	{ c, $ }
S	{ a, c, d }	{ $, c }
Bc	{ a, c }	
DB	{ d, a, c }	
ab	{ a }	
cS	{ c }	
D	{ d }	
E	{**ε** }	

PARSE TABLE

	a	b	c	d	$
S	Bc DB		Bc DB	DB	
B					
D	**ε**		**ε**		

Finish Filling In Table

1. $S \rightarrow AB$
2. $A \rightarrow XY$
3. $A \rightarrow ZW$
4. $X \rightarrow xX$
5. $X \rightarrow \epsilon$
6. $Y \rightarrow yY$
7. $Y \rightarrow \epsilon$
8. $Z \rightarrow zZ$
9. $Z \rightarrow \epsilon$
10. $W \rightarrow wW$
11. $W \rightarrow c$
12. $B \rightarrow bB$
13. $B \rightarrow d$

(a) Grammar

	First	*Follow*
S	z, w, c, x, y, b, d	\$
A	z, w, c, ϵ, x, y	b, d
X	x, ϵ	y, b, d
Y	y, ϵ	b, d
Z	z, ϵ	w, c
W	w, c	b, d
B	b, d	\$

(b) First and Follow sets

Nonter-minal	*Input Symbol*							
	b	c	d	w	x	y	z	\$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	
A	$A \rightarrow XY$	$A \rightarrow ZW$	$A \rightarrow XY$	$A \rightarrow ZW$	$A \rightarrow XY$	$A \rightarrow XY$	$A \rightarrow ZW$	
X	$X \rightarrow \epsilon$		$X \rightarrow \epsilon$		$X \rightarrow xX$	$X \rightarrow \epsilon$		
Y	$Y \rightarrow \epsilon$		$Y \rightarrow \epsilon$			$Y \rightarrow yY$		
Z		$Z \rightarrow \epsilon$		$Z \rightarrow \epsilon$			$Z \rightarrow zZ$	
W		$W \rightarrow c$		$W \rightarrow wW$				
B	$B \rightarrow bB$		$B \rightarrow d$					

examples of ll(1) parser table construction

Question1 - construct ll(1) table of following

E -> TX

T-> (E) | intY

X-> +E| ε

y -> *T|ε

solution -

step1- calculate the first of every token

first(E): {(,int}

first(TX): {(,int}

first(X): {+,ε}

first(Y): {*,ε}

first((E)): {(}

first(intY): {int}

first(+E): {+}

first(*T): {*}

first(ε): {ε}

step2 -calculate the foloow of every terminal

follow(T): {+,),$}

follow(E): {),$}

follow(X): {),$}

follow(Y): {+,),$}

step3- form the tάble

	Int	*	+	(	)	$
E	TX		TX			
T	intY		(E)			
X		+E				ε
Y			ε			ε

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

CONFLICT!

Two rules in one table entry.

i b t i b t o e o

FIRST(S) = { i, o } FOLLOW(S) = { e, $ }

FIRST(S') = { e, ε } FOLLOW(S') = { e, $ }

FIRST(E) = { b } FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'			S' → e S S' → ε			S' → ε
E		E → b				

Algorithm to Build the Table

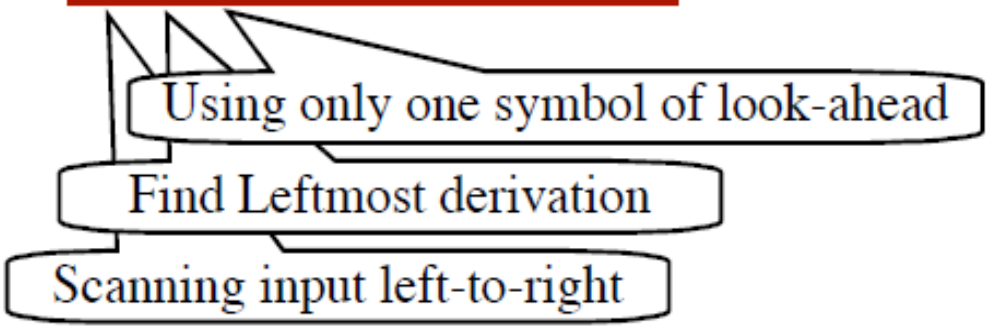
Input: Grammar G

Output: Parsing Table, such that **TABLE[**A,b**]** = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
for each rule A→α do
  for each terminal b in FIRST(α) do
    add A→α to TABLE[A,b]
  endFor
  if ε is in FIRST(α) then
    for each terminal b in FOLLOW(A) do
      add A→α to TABLE[A,b]
    endFor
    if $ is in FOLLOW(A) then
      add A→α to TABLE[A,$]
    endIf
  endIf
endFor
TABLE[A,b] is undefined?  Then set TABLE[A,b] to "error"
TABLE[A,b] is multiply defined?
```

The algorithm fails!!! Grammar G is not LL(1)!!!

LL(1) Grammars



LL(1) grammars

- Are never ambiguous.
- Will never have left recursion.

Furthermore...

If we are looking for an "A" and the next symbol is "b",

Then only one production must be possible.

More Precisely...

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two rules

If $\alpha \Rightarrow^* a\ldots$ and $\beta \Rightarrow^* b\ldots$

then we require $a \neq b$

(i.e., FIRST(α) and FIRST(β) must not intersect)

If $\alpha \Rightarrow^* \varepsilon$

then $\beta \Rightarrow^* \varepsilon$ must not be possible.

(i.e., only one alternative can derive ε.)

If $\alpha \Rightarrow^* \varepsilon$ and $\beta \Rightarrow^* b\ldots$

then b must not be in FOLLOW(A)

LL(1) PARSER

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (i.e. only $ left in the stack), the parsing is completed.

LL(1) PARSER

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol & the special symbol $
- each entry holds a production rule.

LL(1) PARSER – PARSER ACTIONS

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are $ ➔ parser halts (successful completion)
2. If X and a are the same terminal symbol then
   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.
3. If **X is a non-terminal**
   ➔ M [X,a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.
4. none of the above ➔ error
   - **all empty entries in the parsing table are errors.**
   - **If X is a terminal symbol different from a, this is also an error case.**

Table-Driven Predictive Parsing Algorithm

Assume that the grammar is LL(1)

i.e., Backtracking will never be needed

Always know which righthand side to choose (with one look-ahead)

- No Left Recursion
- Grammar is Left-Factored.

Example

$E \rightarrow T\ E'$

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

$T \rightarrow F\ T'$

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

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

Term ...

+Term +Term + ... +Term

Factor ...

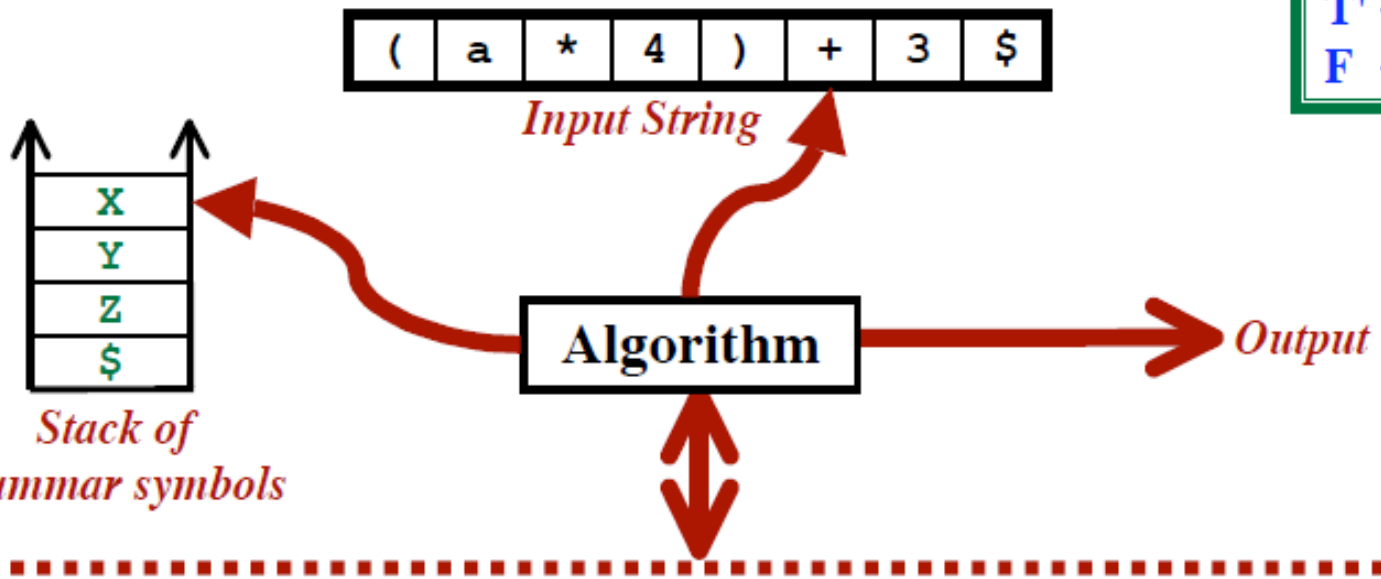
* Factor * Factor * ... * Factor

Step 1: From grammar, construct table.

Step 2: Use table to parse strings.

Table-Driven Predictive Parsing Algorithm

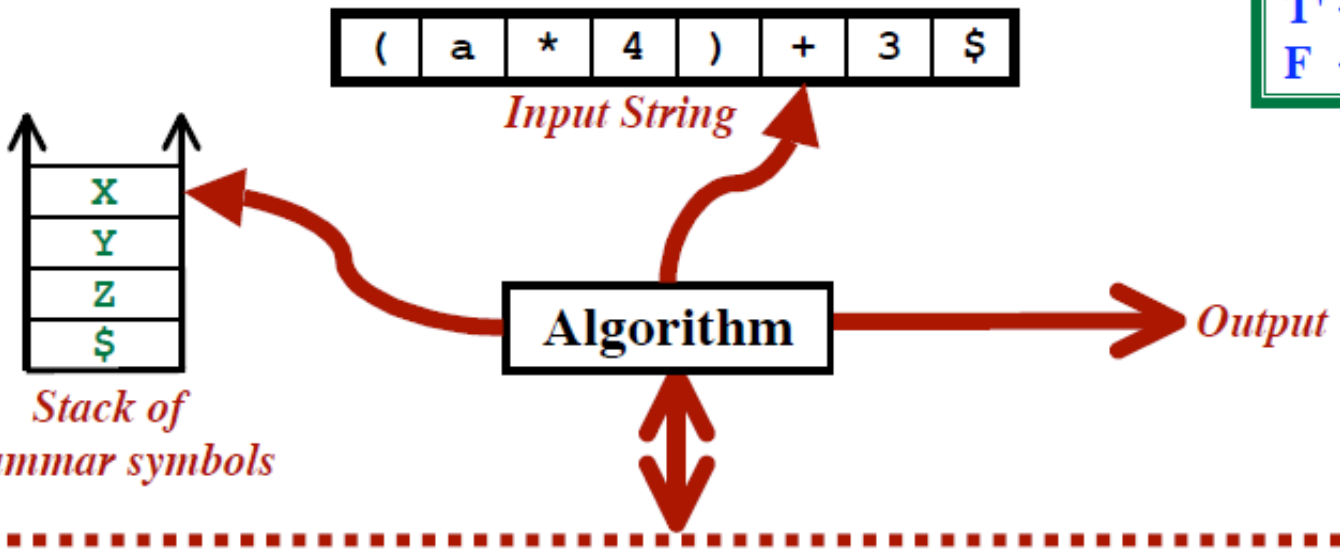
$E \rightarrow T\ E$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$



Pre-Computed Table:

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Table-Driven Predictive Parsing Algorithm



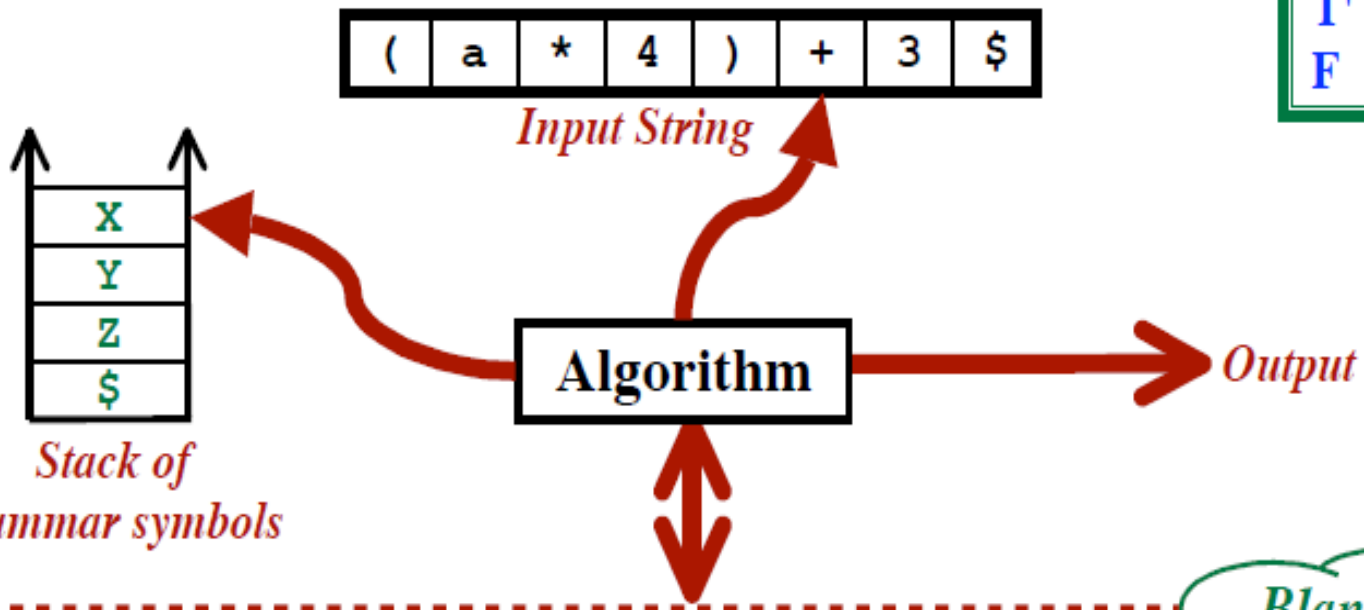
Pre-Computed Table:

Input Symbols, plus $

Nonterminals	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Table-Driven Predictive Parsing Algorithm



Pre-Computed Table:

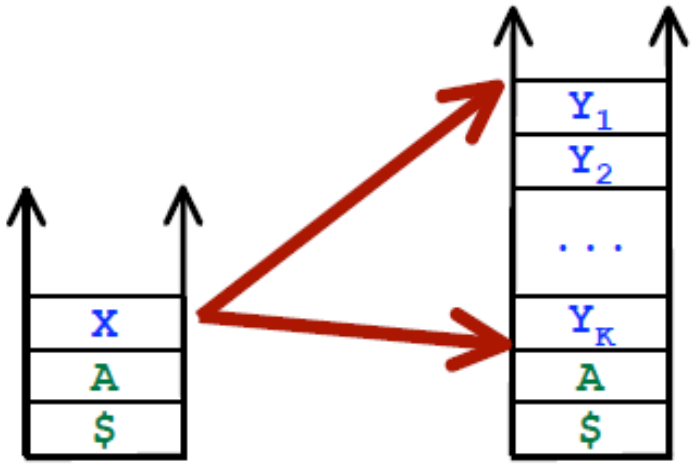
Input Symbols, plus $

Blank entries indicate ERROR

Nonterminals	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing Algorithm

```
Set input ptr to first symbol; Place $ after last input symbol
Push $
Push S
repeat
  X = stack top
  a = current input symbol
  if X is a terminal or X = $ then
    if X == a then
      Pop stack
      Advance input ptr
    else
      Error
    endIf
  elseIf Table[X,a] contains a rule then   // call it X → Y1 Y2 ... YK
    Pop stack
    Push YK
    ...
    Push Y2
    Push Y1
    Print ("X → Y1 Y2 ... YK")
  else   // Table[X,a] is blank
    Syntax Error
  endIf
until X == $
```



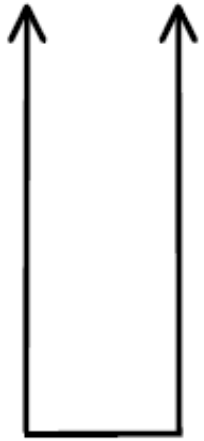
Example

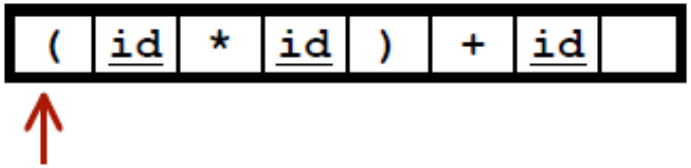
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:





	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

(id*id)+id

Output:

(	id	*	id	)	+	id	

Add $ to end of input
Push $
Push E

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

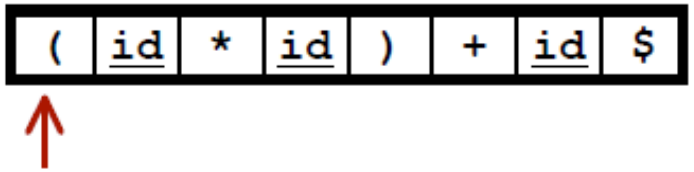
Example

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

`(id*id)+id`

Output:



Add $ to end of input
Push $
Push E

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

Input:

(id*id)+id

Output:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Look at Table [E, '(']
Use rule E→TE'
Pop E
Push E'
Push T
Print E→TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

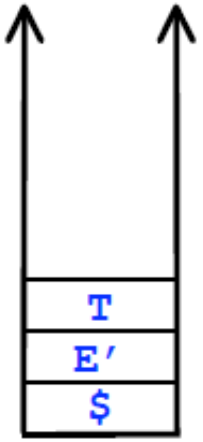
Example

Input:

(id*id)+id

Output:

E → T E'



(	id	*	id	)	+	id	$

Look at Table [E, '(']
Use rule E→TE'
Pop E
Push E'
Push T
Print E→TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

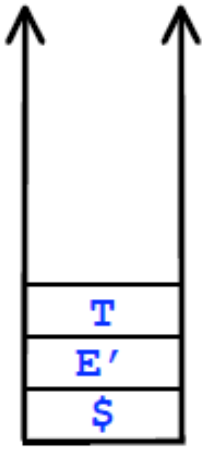
Example

Input:

(id*id)+id

Output:

E → T E'



(	id	*	id	)	+	id	$

Table [T, '('] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

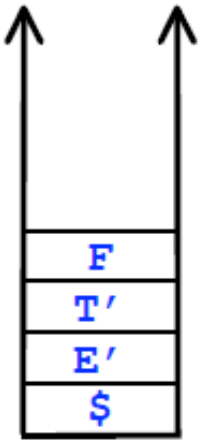
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'



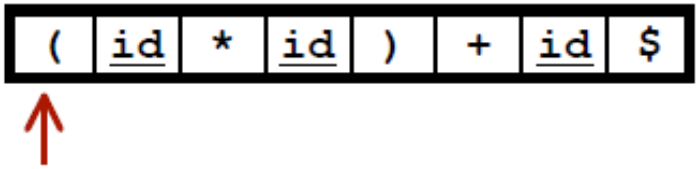


Table [T, '('] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

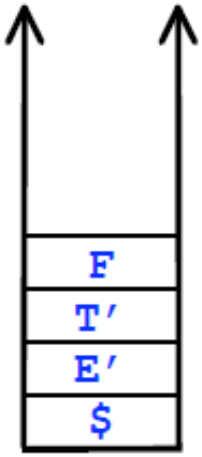
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'



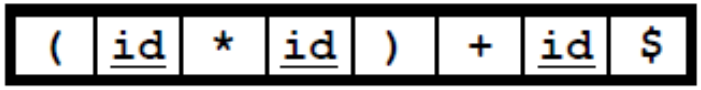


Table [F, '('] = F→(E)
Pop F
Push (
Push E
Push)
Print F→(E)

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

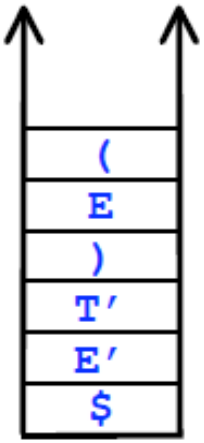
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$

Table [F, '('] = F→(E)
Pop F
Push)
Push E
Push (
Print F→(E)

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

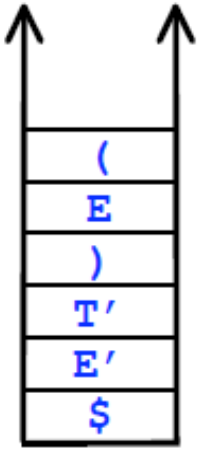
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

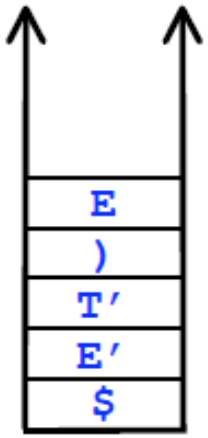
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

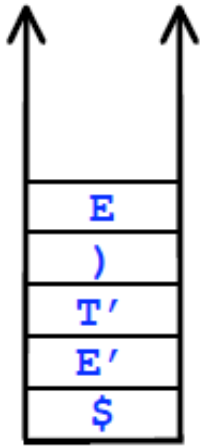
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$

Table [E, id] = E→TE'
Pop E
Push E'
Push T
Print E→TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

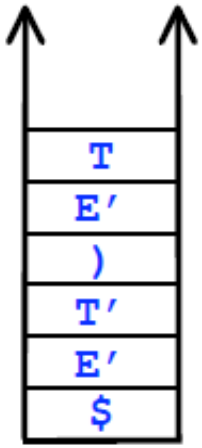
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'



(	id	*	id	)	+	id	$

Table [E, id] = E→TE'
Pop E
Push E'
Push T
Print E→TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

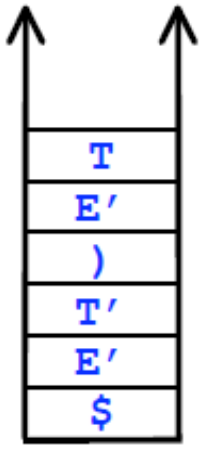
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

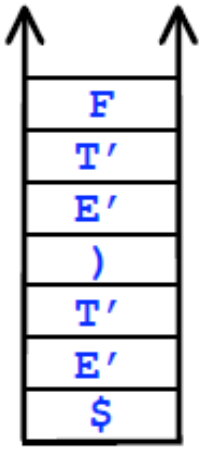
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

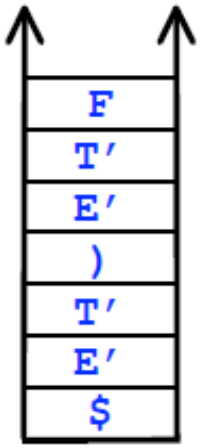
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

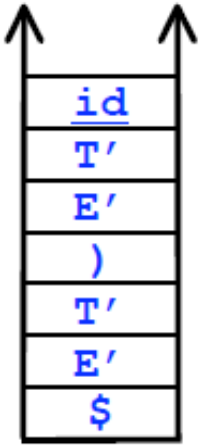
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id



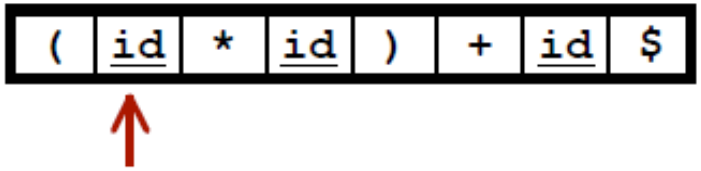


Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

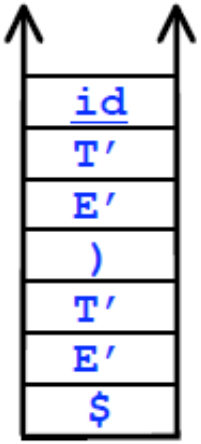
Example

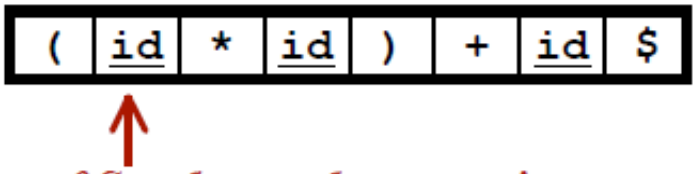
Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id





Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

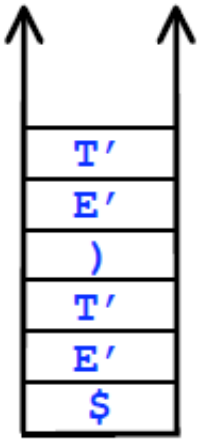
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

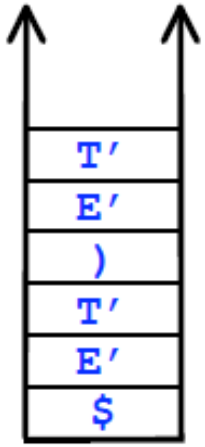
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Table [T', ''] = T'→*FT'*
Pop T'
Push T'
Push F
Push ''*
*Print T'→*FT'*

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

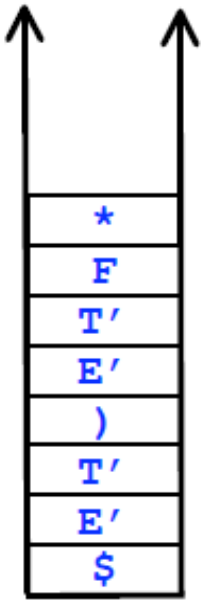
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'



(	id	*	id	)	+	id	$

Table [T', ''] = T'→*FT'*
Pop T'
Push T'
Push F
Push ''*
*Print T'→*FT'*

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

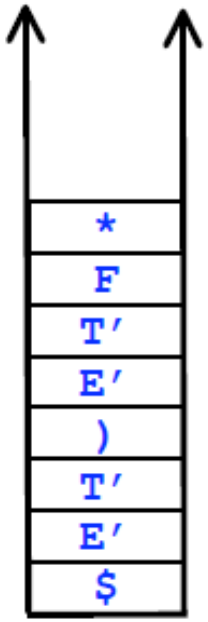
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

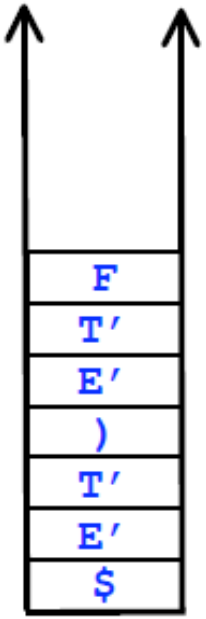
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

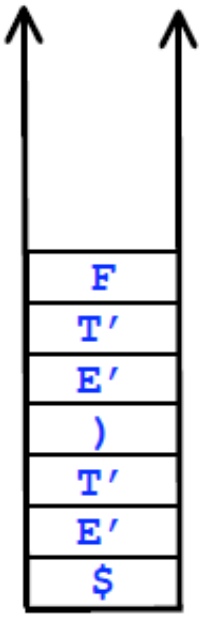
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

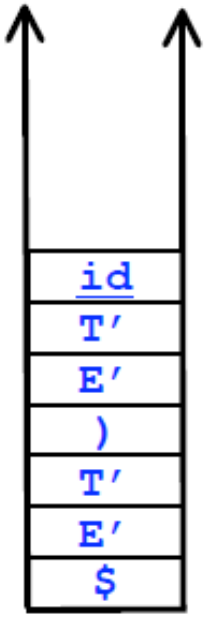
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

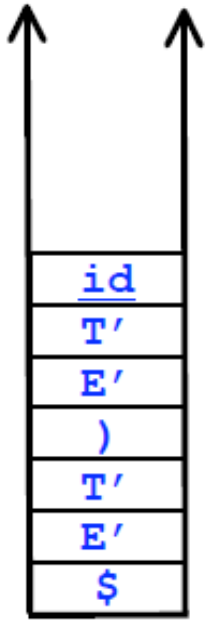
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

`(id*id)+id`

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$
$T' \rightarrow *\ F\ T'$
$F \rightarrow id$



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

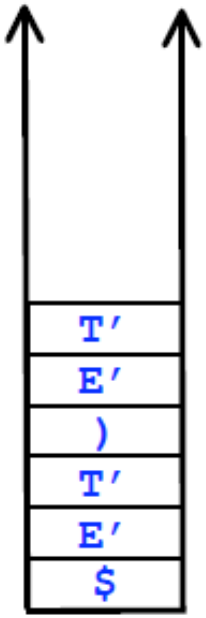
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

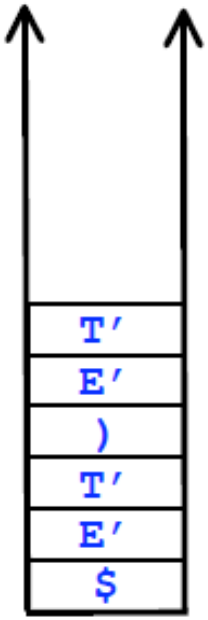
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



(	id	*	id	)	+	id	$

Table [T', ')'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

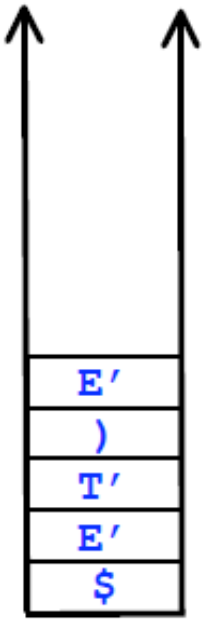
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε



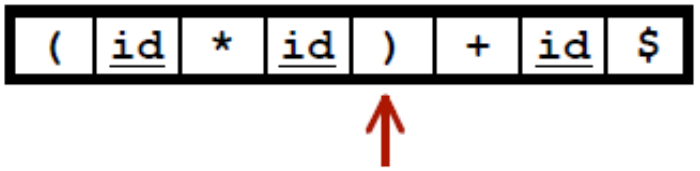


Table [T', ')'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

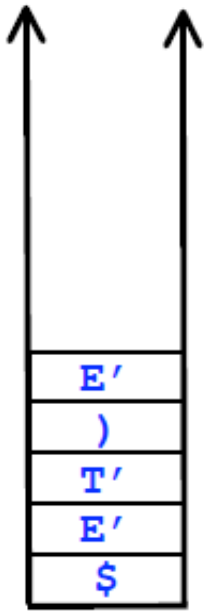
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε



(	id	*	id	)	+	id	$

Table [E', ')'] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

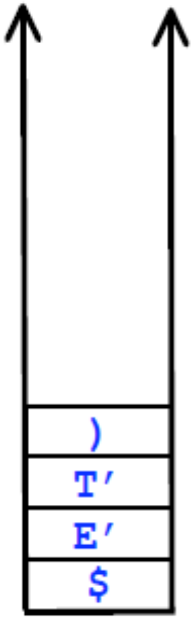
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Table [E', ')'] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

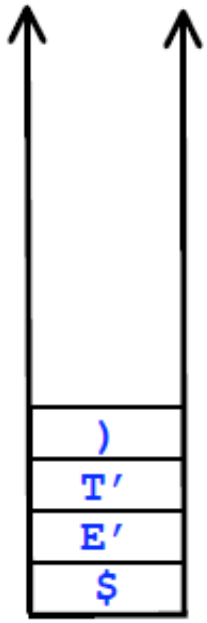
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

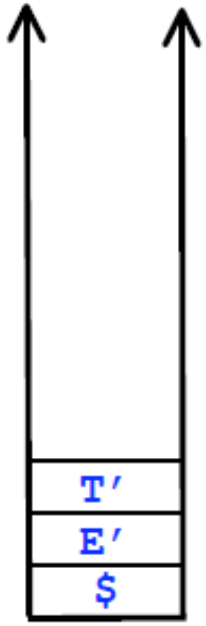
Example

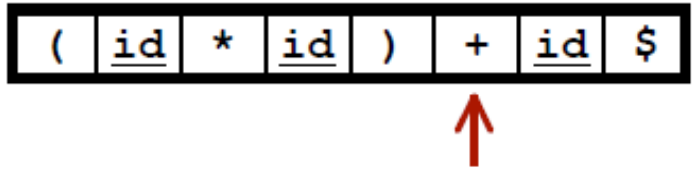
Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε





Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

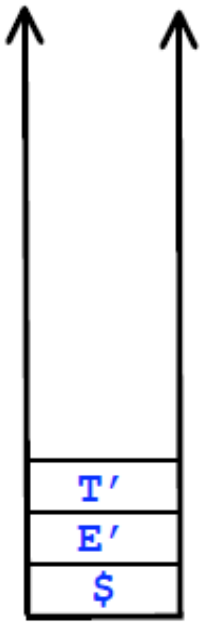
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Table [T', '+'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

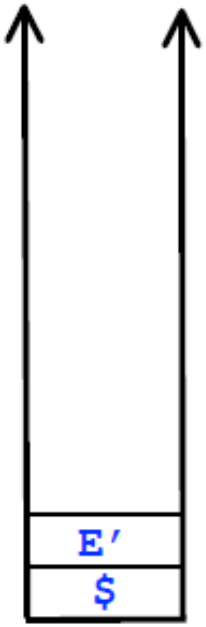
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε



(	id	*	id	)	+	id	$

Table [T', '+'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

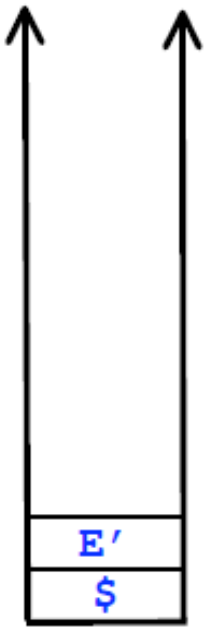
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε



(	id	*	id	)	+	id	$

Table [E', '+'] = E'→+TE'
Pop E'
Push E'
Push T
Push '+'
Print E'→+TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

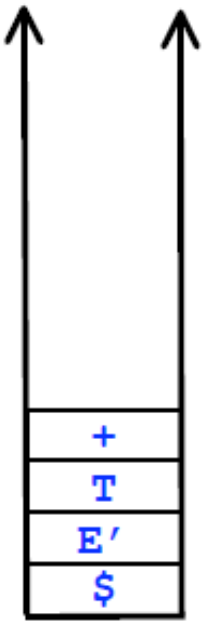
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Table [E', '+'] = E'→+TE'
Pop E'
Push E'
Push T
Push '+'
Print E'→+TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

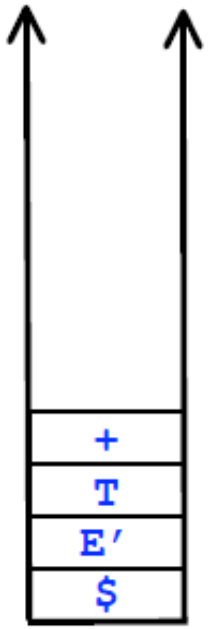
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

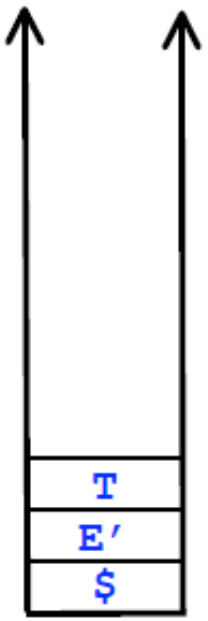
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

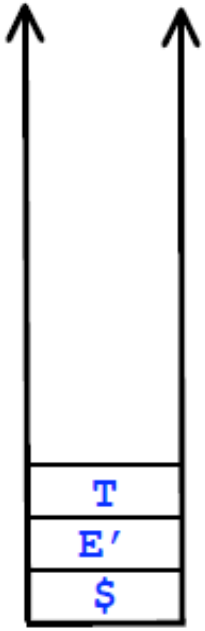
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

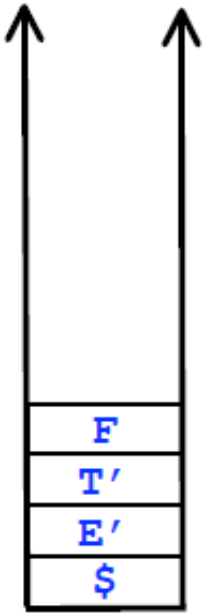
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

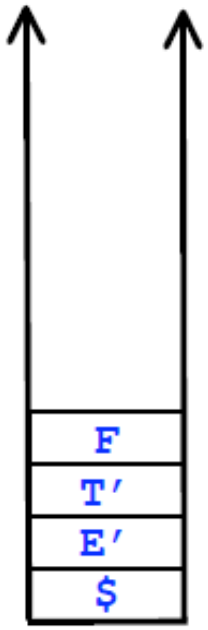
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

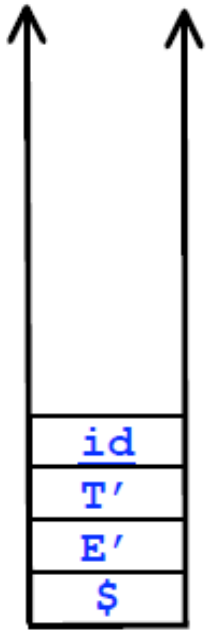
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

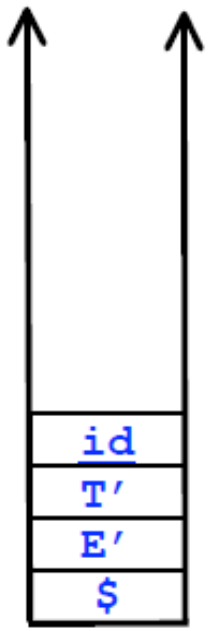
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

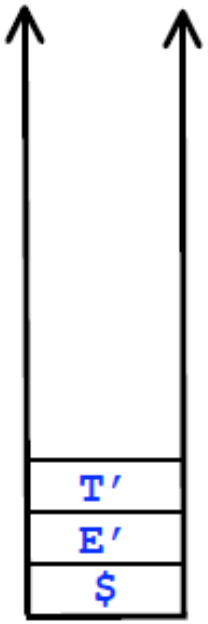
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

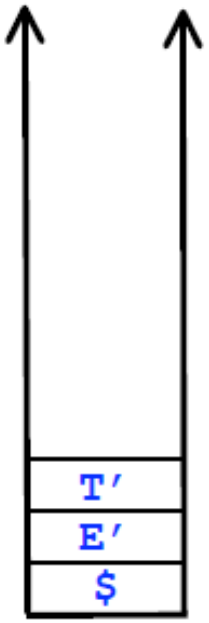
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Table [T', $] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

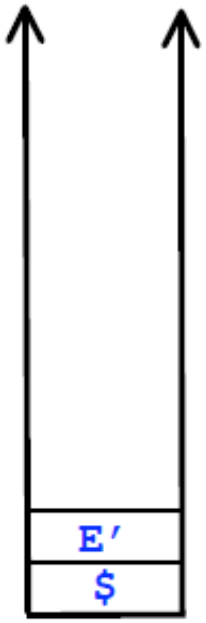
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε



(	id	*	id	)	+	id	$

Table [T', $] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

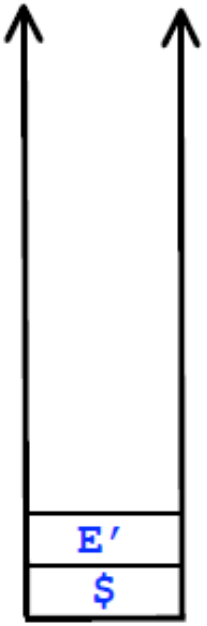
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε



(	id	*	id	)	+	id	$

Table [E', $] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

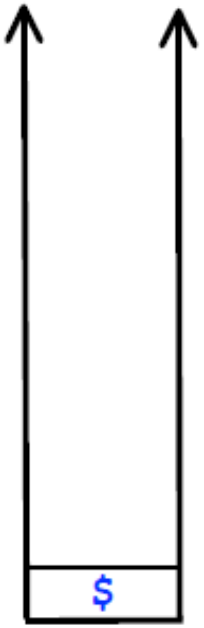
$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \underline{id}
\end{aligned}
$$

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Table [E', $] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Example

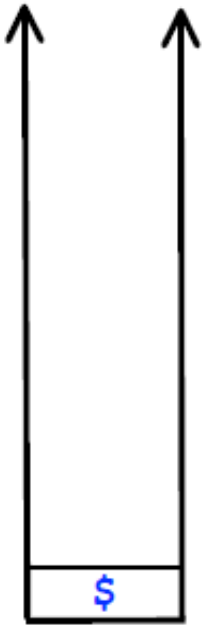
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Input symbol == $
Top of stack == $
Loop terminates with success

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

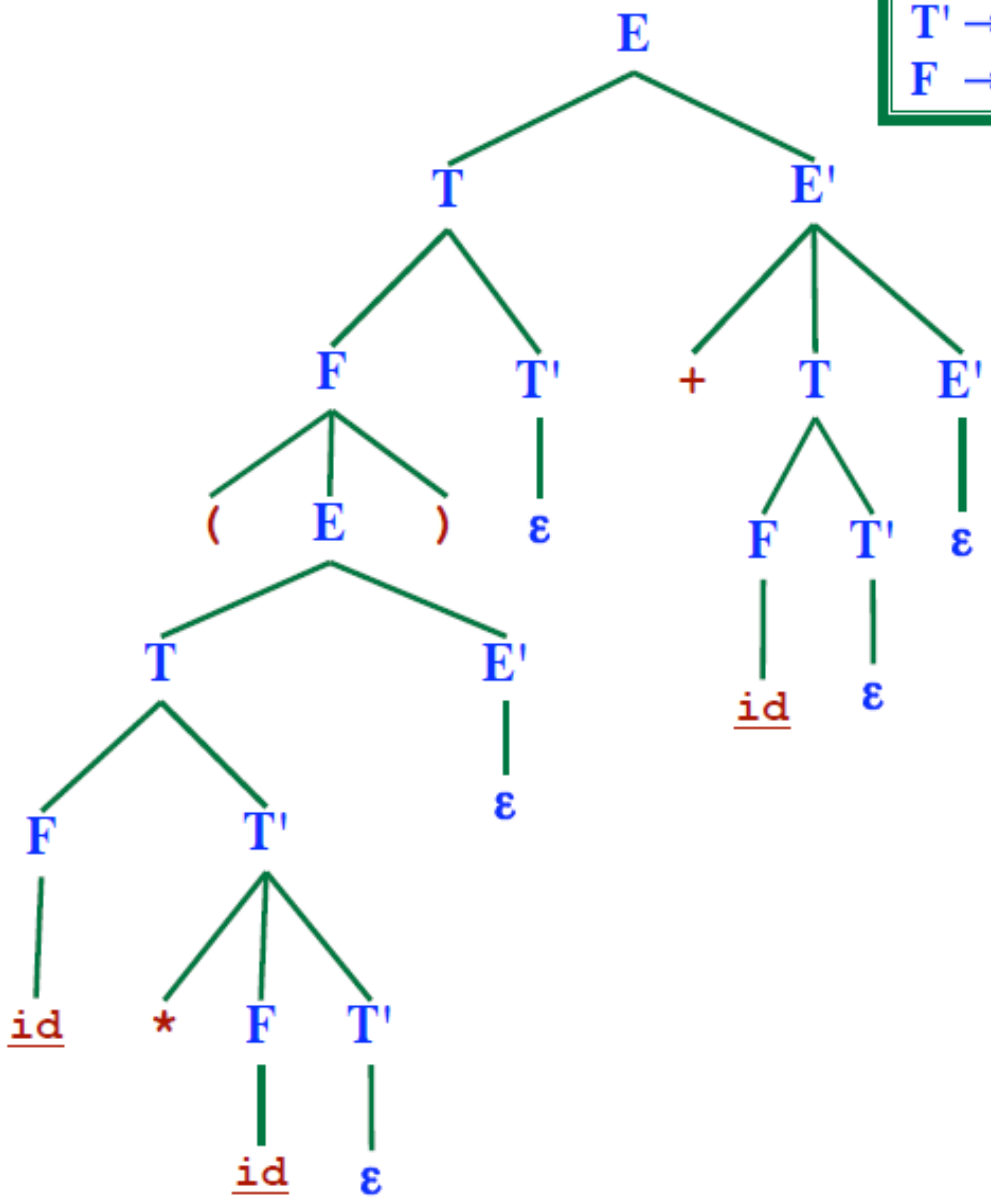
Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε
E'	→	+ T E'
T	→	F T'
F	→	id
T'	→	ε
E'	→	ε



Reconstructing the Parse Tree

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε

Leftmost Derivation:

E
T E'
F T' E'
(E) T' E'
(T E') T' E'
(F T' E') T' E'
(id T' E') T' E'
(id * F T' E') T' E'
(id * id T' E') T' E'
(id * id E') T' E'
(id * id) T' E'
(id * id) E'
(id * id) + T E'
(id * id) + F T' E'
(id * id) + id T' E'
(id * id) + id E'
(id * id) + id

LL(1) PARSER
EXAMPLE TO PARSE ID+ID

stack	input	output
$E	id+id$	E → TE'
$E'T	id+id$	T → FT'
$E'T'F	id+id$	F → id
$ E'T'id	id+id$	
$ E'T'	+id$	T' → ε
$ E'	+id$	E' → +TE'
$ E'T+	+id$	
$ E'T	id$	T → FT'
$ E'T'F	id$	F → id
$ E'T'id	id$	
$ E'T'	$	T' → ε
$ E'	$	E' → ε
$	$	accept

	id	+	$
E	E → TE'		
E'		E' → +TE'	E' → ε
T	T → FT'		
T'		T' → ε	T' → ε
F	F → id		

LL(1) PARSER – ANOTHER EXAMPLE

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

w =abba

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

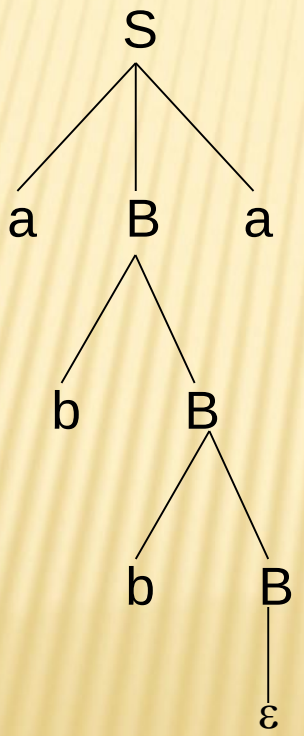
LL(1) Parsing Table

stack	input	output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) PARSER – ANOTHER EXAMPLE (CONT.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

RECURSIVE DESCENT PREDICTIVE PARSING

RECURSIVE DESCENT PREDICTIVE PARSING

PROGRAM → begin DECLIST comma STATELIST end
DECLIS → d semi DECLIST
DECLIST → d
STATELIST → s semi STATELIST
STATELIST → s

After left factoring, the grammer is changed to

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

First(X) = {semi, ϵ} Follow(X) = {comma}

First(Y) = {semi, ϵ} Follow(Y) = {end}

Write functions for each nonterminal.

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

```
main()
{
    token = lexical();
    PROGRAM();
}
```

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

```
Viod PROGRAM
{
     if (token != begin) error();
    token = lexical();
    DECLIST();
    if (token != comma) error();
     token = lexical();
    STATELIST();
    if (token != end) error();
}
```

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

```
void DECLIST()
{
    if (token != d) error;
    token = lexical();
    X();
}
```

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

```
void X()
{
    if (token == semi)
    {
        token = lexical();
        DECLIST();
    }
    else
        if (token == comma) ;        // do nothing
        else error();
}
```

```
void STATELIST()
{
    if (token != s) error();
    token = lexical();
    Y();
}

Void Y()
{
    if (token == semi)
    {
        token = lexical();
        STATELIST();
    }
    else
        if (token == end) ;        // do nothing
        else error();
}
```

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

CHANGING RECURSION INTO ITERATION

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → dX
X → semi DECLIST | ϵ
STATELIST → sY
Y → semi STATELIST | ϵ

Change productions into an extended notation that includes the *.

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → d (semi d)*
STATELIST → s (semi s)*

CHANGING RECURSION INTO ITERATION

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → d (semi d)*
STATELIST → s (semi s)*

```
void DECLIST()
{   if (token != d) error();
    token = lexical();
    while (token == semi)
    {
        token = lexical();
        if (token != d) error();
        token = lexical();
    }
}
```

CHANGING RECURSION INTO ITERATION

PROGRAM → begin DECLIST comma STATELIST end
DECLIST → d (semi d)*
STATELIST → s (semi s)*

```
void STATELIST()
{   if (token != s) error();
    token = lexical();
    while (token == semi)
    {
        token = lexical();
        if (token != s) error();
        token = lexical();
    }
}
```

CHANGING RECURSION INTO ITERATION

Removal of recursion is not always possible. A context free grammar might contain middle recursion and this can not be replaced by iteration. For example

E → E '+' T
E → T
T → T '*' F
T → F
F → '(' E ')'
F → 'x'

E → E '+' T
E → T
T → T '*' F
T → F
F → '(' E ')'
F → 'x'

Transforming the grammar into LL(1)

E → TX
X → ' +' TX | ϵ
T → FY
Y → '*' FY | ϵ
F → '(' E ') | 'x'

Replacing recursion by iteration, where possible, we have

E → T('+' T)*
T → F('*' F)*
F → '(' E ')' | 'x'

```
void E()
{
    T();
    while (token == plus)
    {
        token = lexical();
        T();
    }
}

Void T()
{
    F();
    while (token == Times)
    {
        token = lexical();
        F();
    }
}
```

E → T('+' T)*
T → F('*' F)*
F → '(' E ')' | 'x'

```
Void F()
{
    if (token == obracket)
    {
        token = lexical();
        E();
        if (token == cbracket)
            token = lexical();
        else
            error();
    }
    else   if (token == x)
            token = lexical();
    else
            error();

}

main()
{
    token = lexical(;
    E();
}
```

$E \rightarrow T(\text{`+'}\ T)^*$

$T \rightarrow F(\text{`*'}\ F)^*$

$F \rightarrow \text{`('}\ E\ \text{`)'} \mid \text{`x'}$