CMPSC 160
Translation of Programming Languages

Lecture 4: Top-down parsing and LL(1) parsing

Recap on Last Lecture

- RE vs. CFG
  - Expressiveness and performance
- Key concepts
  - Parse tree, Derivation, Leftmost/rightmost derivation, Sentential form

- Ambiguity: if there is multiple parse tree for one input, then there is ambiguity in our CFG grammar.
  - Break the tie between different choices: else-dangling example
  - Clarify Precedence and Associativity
- Existence: If there is a parse tree, there must be a leftmost and a rightmost derivation.

- But we have not yet talked about how to find/build such a parse tree efficiently? Two general parsing techniques:
  - **Top-down parses + Predicative LL parsers**
  - Bottle-up parses

Top-down Parsing Algorithm

Construct the root node of the parse tree, label it with the start symbol, and set the current-node to root node

Repeat until all the input is consumed (i.e., until the frontier of the parse tree matches the input string)

1 If the label of the current node is a non-terminal node A, **select a (random) production** with A on its lhs and, for each symbol on its rhs, construct the appropriate child (**not terminating**)

2 If the current node is a terminal symbol:

  If it matches the input string, consume it (advance the input pointer)

  If it does not match the input string, **backtrack**[1]

3 Set the current node to the next node in the frontier of the parse tree

  If there is no node left in the frontier of the parse tree and input is not consumed, then **backtrack**[2]

Performance issue: two sources of backtracking + one concern of just not terminating

Example

Let's use the expression grammar with correct precedence and associativity as an example

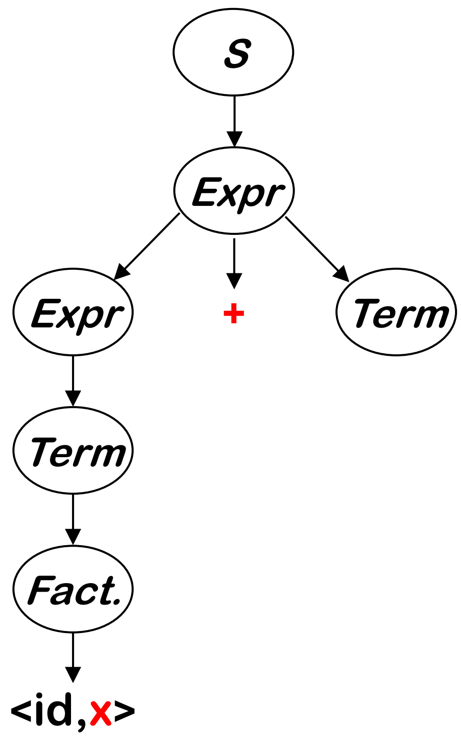
1	*S*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	num
9		|	id

And the input: x – 2 * y

Example: backtrack[1]

Let's try x – 2 * y :

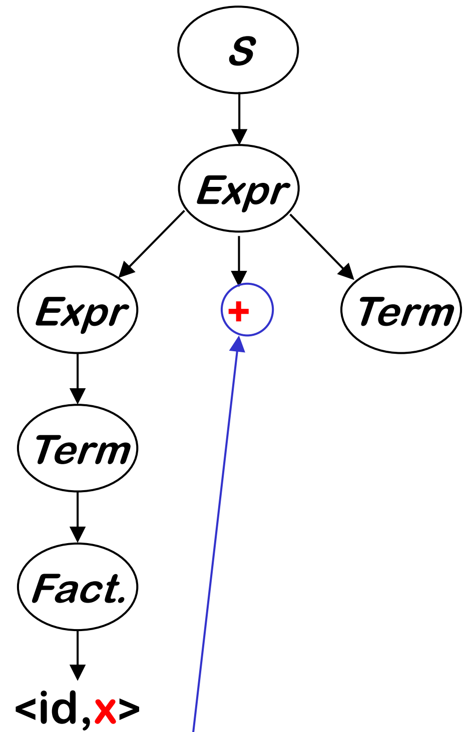
Rule	Sentential Form	Input
-	*S*	↑ x – 2 * y
1	*Expr*	↑ x – 2 * y
2	*Expr* + *Term*	↑ x – 2 * y
4	*Term* + *Term*	↑ x – 2 * y
7	*Factor* + *Term*	↑ x – 2 * y
9	<id,x> + *Term*	↑ x – 2 * y
	<id,x> + *Term*	x ↑ – 2 * y



Example: backtrack[1]

Let's try x – 2 * y :

Rule	Sentential Form	Input
-	*S*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
4	*Term* + *Term*	↑x – 2 * y
7	*Factor* + *Term*	↑x – 2 * y
9	<id,x> + *Term*	↑x – 2 * y
	<id,x> + *Term*	x ↑– 2 * y



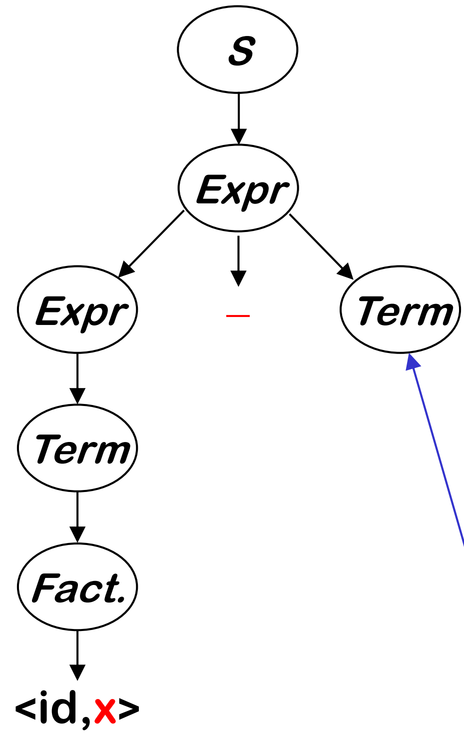
Note that "–" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
-	*S*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr – Term*	↑x – 2 * y
4	*Term – Term*	↑x – 2 * y
7	*Factor – Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
-	<id,x> – *Term*	x ↑– 2 * y
-	<id,x> – *Term*	x – ↑2 * y



This time "–" and "–" matched

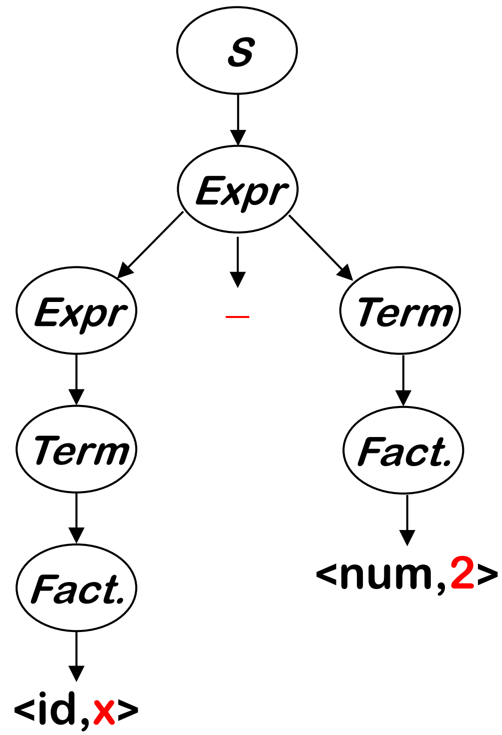
We can advance past "–" to look at "2"

Now we need to extend *Term,* the last *NT* in the fringe of the parse tree

Example: backtrack^2

Trying to match the “2” in x – 2 * y :

Rule	Sentential Form	Input
–	<id,x> – *Term*	x – ↑2 * y
7	<id,x> – *Factor*	x – ↑2 * y
9	<id,x> – <num,2>	x – ↑2 * y
–	<id,x> – <num,2>	x – 2 ↑* y



Where are we?

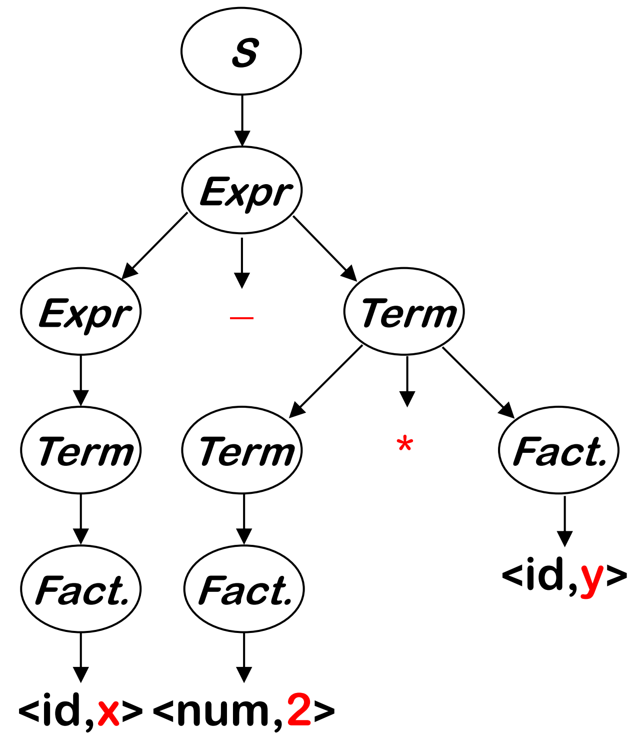
- num matches “2”
- We have more input, but no *NT*s left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example: backtrack[2]

Trying again with “2” in x – 2 * y :

Rule	Sentential Form	Input
-	<id,x> – *Term*	x – ↑2 * y
5	<id,x> – *Term* * *Factor*	x – ↑2 * y
7	<id,x> – *Factor* * *Factor*	x – ↑2 * y
8	<id,x> – <num,2> * *Factor*	x – ↑2 * y
-	<id,x> – <num,2> * *Factor*	x – 2 ↑* y
-	<id,x> – <num,2> * *Factor*	x – 2 * ↑y
9	<id,x> – <num,2> * <id,y>	x – 2 * ↑y
-	<id,x> – <num,2> * <id,y>	x – 2 * y ↑



This time, we matched and consumed all the input

⇒ Success!

Another possible parsing

Other choices for expansion are possible

Rule	*Sentential Form*	*Input*
—	*S*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term* + ...+ *Term*	↑x - 2 * y

consuming no input !

This does not terminate

- Wrong choice of expansion leads to non-termination, the parser will not backtrack since it does not get to a point where it can backtrack
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Left Recursion

Top-down parsers cannot handle ***left-recursive grammars***

Our expression grammar is left-recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion (without changing the language that is defined by the grammar)

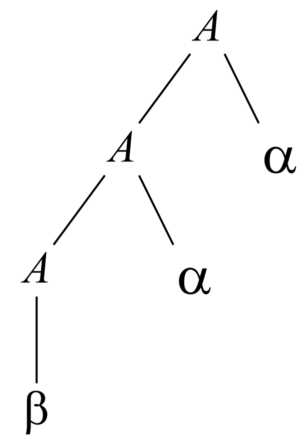
A grammar is *left recursive* if there exists a non-terminal A such that there exists a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

$\Rightarrow^{+}$ means a bunch of (1 or more) productions applied in series

$\alpha \in (NT \cup T)^{+}$ means that α is a non-empty sequence of nonterminal and terminal symbols

Eliminating Immediate Left Recursion

To remove left recursion, we can transform the grammar



Consider a grammar fragment of the form

$$A \rightarrow A\ \alpha$$
$$\quad |\ \ \beta$$

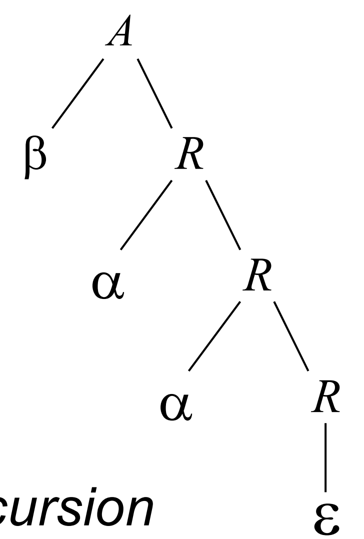
where α or β are strings of terminal and non-terminal symbols

and neither α nor β start with *A*

We can rewrite this as

$$A \rightarrow \beta\ R$$
$$R \rightarrow \alpha\ R$$
$$\quad |\ \ \varepsilon$$

where *R* is a new non-terminal



This accepts the same language, but uses only right recursion

Question Time ☺

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots . | A\alpha_m | \beta_1 | \beta_2 | \ldots . . \beta_n \quad \Longrightarrow \quad ?$$

Given

$$\begin{aligned} A &\rightarrow A\,\alpha \\ &\mid\ \beta \end{aligned} \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow \beta\, R \\ R &\rightarrow \alpha\, R \\ &\mid\ \varepsilon \end{aligned}$$

Eliminating Immediate Left Recursion

- The general form for left recursion is

$$A \rightarrow A\alpha_1|A\alpha_2| \ldots . |A\alpha_m|\beta_1|\beta_2| \ldots . . \beta_n$$

is can be replaced by

$$A \rightarrow \beta_1A'|\beta_2A'| \ldots . . | \ldots . . |\beta_nA'$$

$$A' \rightarrow \alpha_1A'|\alpha_2A'| \ldots . . |\alpha_mA'|\varepsilon$$

Is the converted CFG much more complicated?

Example

The expression grammar contains two cases of left recursion

Expr	$\rightarrow$	***Expr + Term***	***Term***	$\rightarrow$	***Term * Factor***
	|	***Expr – Term***		|	***Term / Factor***
	|	***Term***		|	***Factor***

Applying the transformation yields

Expr	$\rightarrow$	***Term Expr′***	***Term***	$\rightarrow$	***Factor Term′***
Expr′	$\rightarrow$	***+ Term Expr′***	***Term′***	$\rightarrow$	**** Factor Term′***
	|	***- Term Expr′***		|	***/ Factor Term′***
	|	ε		|	ε

$$A \rightarrow A\alpha_1|A\alpha_2| \ldots . |A\alpha_m|\beta_1|\beta_2| \ldots . . \beta_n$$

$$\longrightarrow$$

$$A \rightarrow \beta_1A'|\beta_2A'| \ldots . . | \ldots . . |\beta_nA'$$

$$A' \rightarrow \alpha_1A'|\alpha_2A'| \ldots . . |\alpha_mA'|\varepsilon$$

Left-Recursive and Right-Recursive Grammar

1	*S*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	$\rightarrow$	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	$\rightarrow$	num
9		|	id

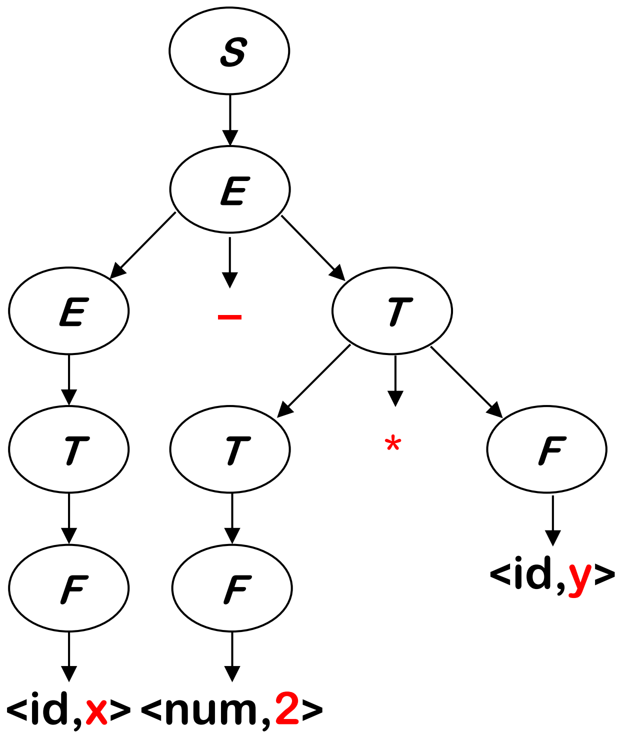
1	*S*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Term Expr′*
3	*Expr′*	$\rightarrow$	+ *Term Expr′*
4		|	– *Term Expr′*
5		|	ε
6	*Term*	$\rightarrow$	*Factor Term′*
7	*Term′*	$\rightarrow$	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	$\rightarrow$	num
11		|	id

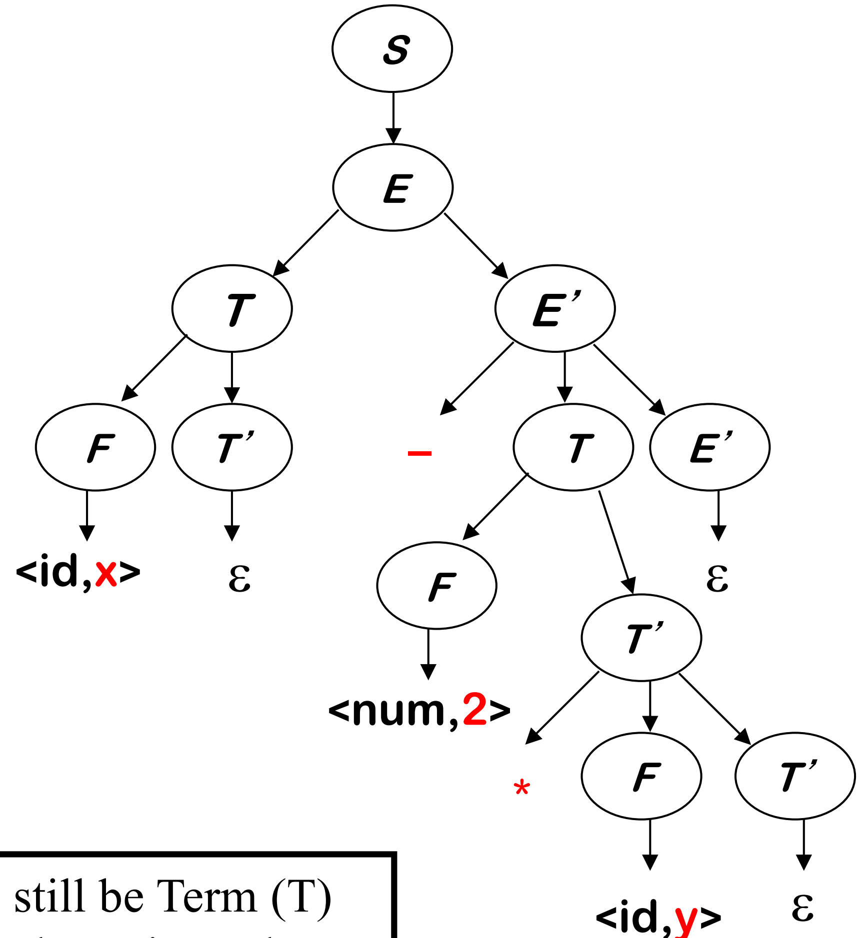
This grammar is correct, if somewhat non-intuitive.

A top-down parser will terminate using it.

Q: Will the transformation change the associativity and precedence?

Preserves Precedence





Key: the root node for a multiplication subtree will still be Term (T) must be a child node of some E, a root node for a subtraction subtree.

Eliminating Indirect Left Recursion

Example:

A -> Cx
B -> Cy
C -> A | B | z

Rule: first establish some kind of order for non-terminals, and then find all paths where indirect recursion happens.

Order: C < B < A

NT can only have NT with lower order on the RHS of its production rule

C -> Cx | Cy | z

↓

C -> zC'
C' -> xC' | yC' | ε

↓

A -> Cx
B -> Cy
C -> zC'
C' -> xC' | yC' | ε

Done!!!

Eliminating Left Recursion

The previous transformation eliminates immediate left recursion

What about more general, indirect left recursion?

The general algorithm (Algorithm 4.1 in the Textbook):

Arrange the NTs into some order $A_1, A_2, \ldots, A_n$

for $i \leftarrow 1$ to n

for $j \leftarrow 1$ to i-1

replace each production $A_i \rightarrow A_j\ \gamma$ with

$A_i \rightarrow \delta_1 \gamma \;/ \delta_2 \gamma / \ldots / \delta_k \gamma,$

where $A_j \rightarrow \delta_1 \;/ \delta_2 / \ldots / \delta_k$ are all the current productions for A_j

eliminate any immediate left recursion on A_i using the direct transformation

Efficiency with Backtracking?

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly.

Solution: **Predictive Parsing**

Basic idea

The main idea is to look ahead at the next few tokens and use that token to pick the production that you should apply

$X \rightarrow + X$	Here we can use the + and –
$\quad \mid \ - Y$	to decide which rule to apply

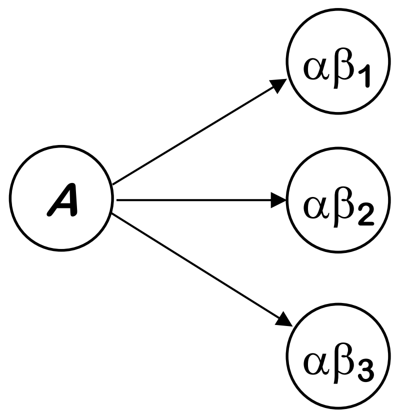
What is the potential problem here?

$A \rightarrow \alpha B \mid \alpha C$

Left Factoring

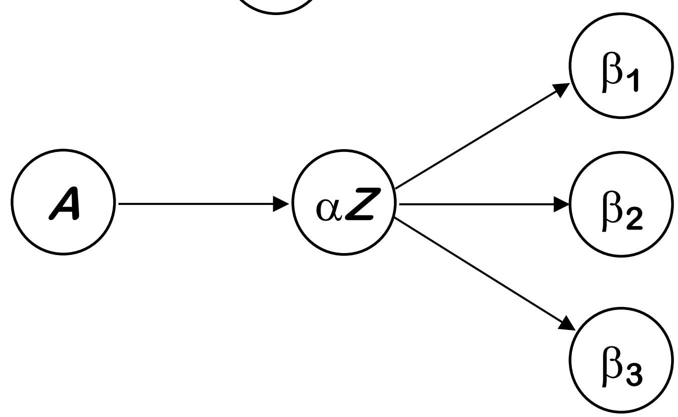
A graphical explanation for the left-factoring

$$\begin{aligned} A &\rightarrow \alpha\beta_1 \\ &\mid \alpha\beta_2 \\ &\mid \alpha\beta_n \end{aligned}$$



becomes …

$$\begin{aligned} A &\rightarrow \alpha\ Z \\ Z &\rightarrow \beta_1 \\ &\mid\ \beta_2 \\ &\mid\ \beta_n \end{aligned}$$



Left Factoring

- We already learned one transformation: Removing left-recursion
- There is another transformation called left-factoring

Left-Factoring Algorithm:

$\forall$ ***A*** $\in$ ***NT,***

find the longest prefix α ***that occurs in two or more right-hand sides of A***

if $\alpha \neq \varepsilon$ ***then replace all of the A productions,***

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k ,$$

with

$$A \rightarrow \alpha\, Z \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$$

$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where Z is a new element of NT

Repeat until no common prefixes remain

More on Predictive Parsing

$A \longrightarrow \alpha 1$
$A \longrightarrow \alpha 2$
$A \longrightarrow \alpha 3$

- Suppose all three rules have some NT on the RHS?
- Which rule to select if the next token is "x"? No idea.

❖ First Set for a Non-Terminal (NT) α :

$x \in \text{FIRST}(\alpha)$ *iff* 1) $\alpha \Rightarrow^* \boldsymbol{x}\, \gamma$, for some $\gamma \in (NT \cup T)^*$ and $x \in T$

2) $\alpha \Rightarrow^* \varepsilon$ and $x = \varepsilon$

ALL tokens that can be at the beginning of a string that can be derived from α

❖ First set for a Terminal (T) x

$x = \text{FIRST}(x)$

$\boxed{\text{It seems that if all first sets are distinct, we are done!}}$

$\text{Predict}(A \longrightarrow \alpha 1) = \text{First}(\alpha 1)$
$\text{Predict}(A \longrightarrow \alpha 2) = \text{First}(\alpha 2)$
$\text{Predict}(A \longrightarrow \alpha 3) = \text{First}(\alpha 3)$

Question Time ☺

$S \rightarrow AB$
$A \rightarrow x \mid y$
$B \rightarrow 0 \mid 1$

FIRST(S) = { }

Definition: ALL terminals that can be at the beginning of a string that can be derived from S.

$S \rightarrow AB$
$A \rightarrow x \mid y \mid \varepsilon$
$B \rightarrow 0 \mid 1$

FIRST(S) = { }

$S \rightarrow AB$
$A \rightarrow x \mid y \mid \varepsilon$
$B \rightarrow 0 \mid 1 \mid \varepsilon$

FIRST(S) = { }

Examples: Compute FIRST(S)

$S \rightarrow AB$
$A \rightarrow x \mid y$
$B \rightarrow 0 \mid 1$

FIRST(S) = { x, y }

$S \rightarrow AB$
$A \rightarrow x \mid y \mid \varepsilon$
$B \rightarrow 0 \mid 1$

FIRST(S) = { x, y, 0, 1 }

$S \rightarrow AB$
$A \rightarrow x \mid y \mid \varepsilon$
$B \rightarrow 0 \mid 1 \mid \varepsilon$

FIRST(S) = { x, y, 0, 1, ε }

How to Compute FIRST Sets

To construct FIRST(X) for a grammar symbol X, apply the following rules until no more symbols can be added to FIRST(X)

1. If X is a terminal, then FIRST(X) = X
2. If X is a non-terminal and $X \rightarrow \varepsilon$ is a production, then put ε in FIRST(X)
3. If X is a non-terminal and $X \rightarrow t$ is a production (t is a terminal), then put t in FIRST(X)
4. If X is a non-terminal and $X \rightarrow Y_1 | Y_2 \ldots | Y_k$ is a production, then let FIRST(X) = FIRST(Y_1) $\cup$ FIRST(Y_2) … $\cup$ FIRST(Y_k)
5. If X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then
   - put every symbol in FIRST(Y_1) other than ε to FIRST(X)
   - if ε is in FIRST(Y_j) for all $1 \leq j < l$, put every symbol in FIRST(Y_l) other than ε to FIRST(X)
   - put ε in FIRST(X) if ε is in FIRST(Y_i) for all $1 \leq i \leq k$

We still have those pesky epsilons …

$T \rightarrow S\ z$

$S \rightarrow AB$

$A \rightarrow x \mid y \mid \varepsilon$

$B \rightarrow 0 \mid 1 \mid \varepsilon$

$\text{FIRST}(S) = \{\ x, y, 0, 1, \varepsilon\ \}$

- Suppose we current node to process is S, and the next token is z. Are we having an error?

- We shall also examine the set of characters that can *follow* the current non-terminal If we have ε in our FIRST sets
- This is what the FOLLOW set defines.
  - FOLLOW(*A*) for a non-terminal symbol *A*.
    - The set of terminal symbols that can appear immediately to the right of *A* in some sentential form.

How to Compute FOLLOW Sets?

To construct FOLLOW(*A*) for a non-terminal symbol *A*, apply the following rules until no more symbols can be added to FOLLOW(*A*):

1. Put \$ in FOLLOW(*S*) (\$ is the end-of-file symbol, *S* is the start symbol)

2. If there is a production $B \to \alpha\ A\ \beta$,
   - then put everything in FIRST(β) - except ε - in FOLLOW(*A*)
   - if ε is in FIRST(β), then put everything in FOLLOW(*B*) in FOLLOW(*A*)

3. If there is a production $B \to \alpha\ A$, then put everything in FOLLOW(*B*) in FOLLOW(*A*)

FIRST/FOLLOW Example

Example Input: x + y (z + a (b))

Expression → Function
| **(** Expression **)**
| Primary **+** Expression
| Primary
Primary → **id**
| **num**
Function → **id (** ParamList **)**
ParamList → Expression ParamList
| ε

- First (α): ALL tokens that can be at the beginning of a string that can be derived from α.
- FOLLOW(α): The set of terminal symbols that can appear immediately to the right of α in some sentential form.

FIRST (Expression) = {}
FIRST (Primary) = {}
FIRST (Function) = { }
FIRST (ParamList) = {}

FOLLOW (Expression) = {}
FOLLOW (Primary) = {}
FOLLOW (Function) = {}
FOLLOW (ParamList) = { }

FIRST/FOLLOW Example

Example Input: x + y (z + a (b))

Expression $\rightarrow$ Function
| **(** Expression **)**
| Primary **+** Expression
| Primary
Primary $\rightarrow$ **id**
| **num**
Function $\rightarrow$ **id (** ParamList **)**
ParamList $\rightarrow$ Expression ParamList
| ε

- First (α): ALL tokens that can be at the beginning of a string that can be derived from α.
- FOLLOW(α): The set of terminal symbols that can appear immediately to the right of α in some sentential form.

FIRST (Expression) = { **(** , **num, id** }
FIRST (Primary) = { **num, id** }
FIRST (Function) = { **id** }
FIRST (ParamList) = { **id** , **num, (** , ε }

FOLLOW (Expression) = { **$** , **(** , **)** , **id, num** }
FOLLOW (Primary) = { **$** , **(** , **)** , **+** , **id, num** }
FOLLOW (Function) = { **$** , **(** , **)** , **id, num** }
FOLLOW (ParamList) = { **)** }

LL(1) Grammars

Left-to-right scan of the input, **L**eftmost derivation, **1**-token look-ahead

A grammar G is LL(1) if for each set of its productions
$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

FIRST(α_1), FIRST(α_2), ..., FIRST(α_n), are all pair-wise disjoint

If $\alpha_i \Rightarrow^* \varepsilon$, then FIRST (α_j) $\cap$ FOLLOW (A) = $\varnothing$ for all $1 \leq i \leq n$, $i \neq j$

- In other words, LL(1) grammars
  - during leftmost derivation, productions are uniquely predictable with a one token lookahead

Where are we in the process?