Parsing — Part II
(Top-down parsing, left-recursion removal)

CS434
Lecture 7
Spring 2005
Department of Computer Science
University of Alabama
Joel Jones

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Parsing Techniques

Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

A top-down parser starts with the root of the parse tree

The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until the fringe of the parse tree matches the input string

1. *At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child*
2. *When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack*
3. *Find the next node to be expanded* (label $\in$ NT)

- The key is picking the right production in step 1
  - → That choice should be guided by the input string

Remember the expression grammar?

Version with precedence derived last lecture

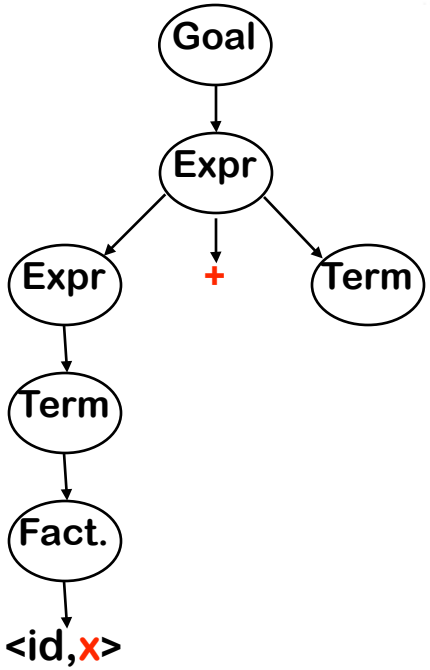
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

And the input x – 2 * y

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑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
9	<id,x> + Term	x ↑– 2 * y

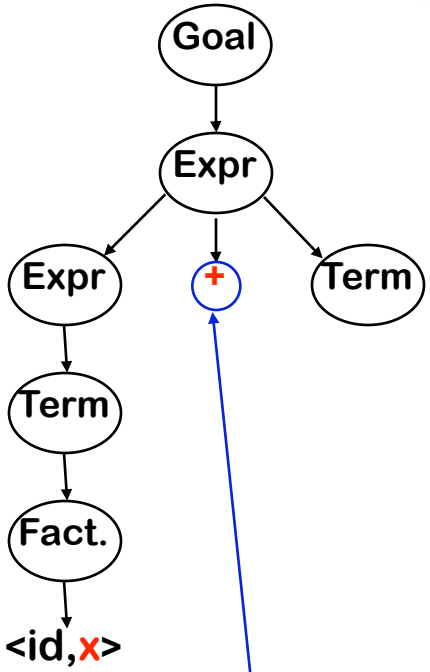


Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	Goal	↑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
9	<id,x> + Term	x ↑- 2 * y



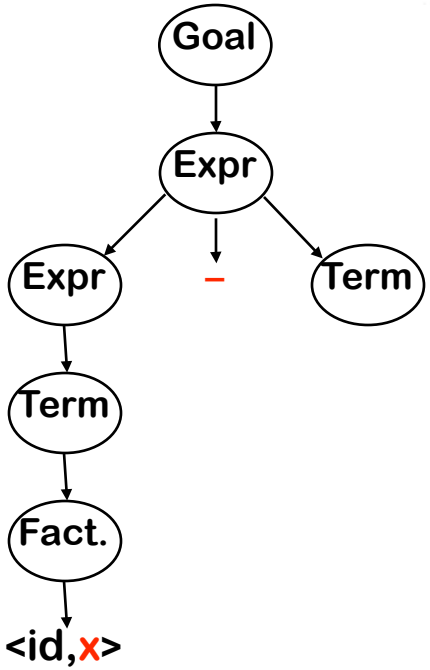
This worked well, except that "-" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x – 2 * y :

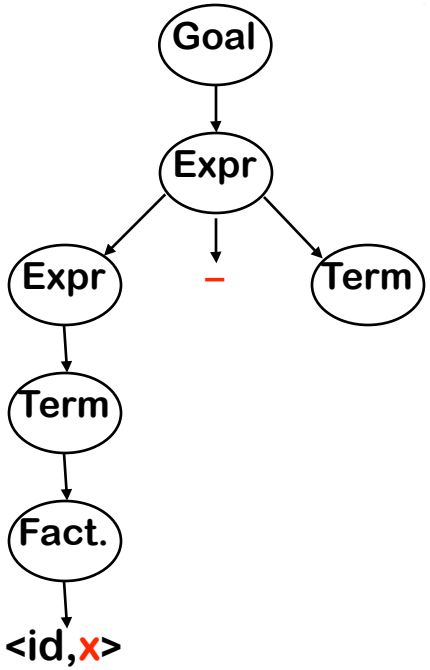
Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr – Term*	↑x – 2 * y
4	*Term – Term*	↑x – 2 * y
7	*Factor – Term*	↑x – 2 * y
9	<id,x> *– Term*	↑x – 2 * y
9	<id,x> *– Term*	x ↑– 2 * y
—	<id,x> *– Term*	x – ↑2 * y



Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑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
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



This time, "–" and "–" matched

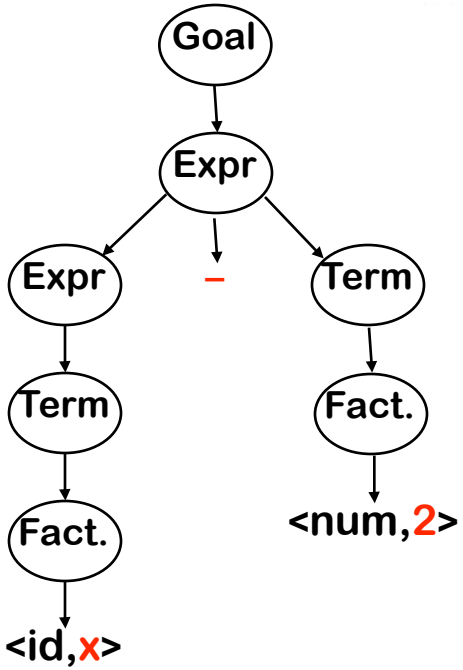
We can advance past "–" to look at "2"

⇒ Now, we need to expand Term - the last NT on the fringe

Example

Trying to match the "2" in x - 2 * y :

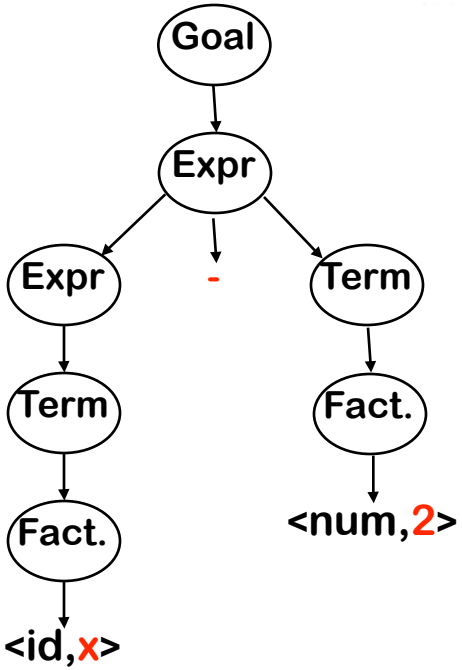
Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y



Example

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?

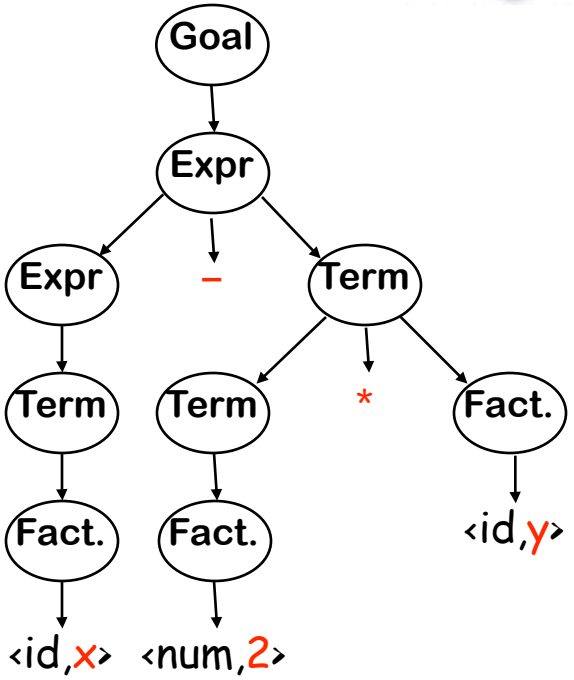
- "2" matches "2"
- We have more input, but no NTs left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

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 & consumed all the input

⇒ Success!

Another possible parse

Other choices for expansion are possible

Rule	Sentential Form	Input
—	*Goal*	↑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 doesn't terminate (obviously)

- Wrong choice of expansion leads to non-termination
- 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

Formally,

A grammar is left recursive if $\exists\ A \in NT$ such that

$\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

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

Non-termination is a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$\text{Fee} \rightarrow \text{Fee}\ \alpha \mid \beta$$

where neither α nor β start with Fee

We can rewrite this as

$$\text{Fee} \rightarrow \beta\ \text{Fie}$$

$$\text{Fie} \rightarrow \alpha\ \text{Fie} \mid \varepsilon$$

where Fie is a new non-terminal

This accepts the same language, but uses only right recursion

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / Factor
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	→	*Term Expr′*	*Term*	→	*Factor Term′*
Expr′	|	+ *Term Expr′*	*Term′*	|	* *Factor Term′*
	|	- *Term Expr′*		|	/ *Factor Term′*
	|	ε		|	ε

These fragments use only right recursion

They retain the original left associativity

Eliminating Left Recursion

Substituting them back into the grammar yields

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.

Eliminating Left Recursion

The transformation eliminates immediate left recursion

What about more general, indirect left recursion ?

The general algorithm:

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

for i ← 1 to n

  for s ← 1 to i - 1

    replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,

    where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for A_s

  eliminate any immediate left recursion on A_i

    using the direct transformation

Must start with 1 to ensure that $A_1 \rightarrow A_1 \beta$ is transformed

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$),

and no epsilon productions

And back

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding A_i has no non-terminal A_s in its rhs, for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion

At the start of the i^{th} outer loop iteration

For all $k < i$, no production that expands A_k contains a non-terminal

A_s in its rhs, for $s < k$

Example

- Order of symbols: G, E, T

G → E

E → E + T

E → T

T → E ~ T

T → id

Example

- Order of symbols: G, E, T

1. A_i = G

G → E

E → E + T

E → T

T → E ~ T

T → <u>id</u>

Example

- Order of symbols: G, E, T

1. A_i = G	2. A_i = E
G → E	G → E
E → E + T	E → T E'
E → T	E' → + T E'
T → E ~ T	E' → ε
T → id	T → E ~ T
	T → id

Example

- Order of symbols: G, E, T

1. A_i = G	2. A_i = E	3. A_i = T, A_S = E
G → E	G → E	G → E
E → E + T	E → T E'	E → T E'
E → T	E' → + T E'	E' → + T E'
T → E ~ T	E' → ε	E' → ε
T → id	T → E ~ T	T → T E' ~ T
	T → id	T → id

Go to Algorithm

Example

- Order of symbols: G, E, T

1. A_i = G	2. A_i = E	3. A_i = T, A_S = E	4. A_i = T
G → E	G → E	G → E	G → E
E → E + T	E → T E'	E → T E'	E → T E'
E → T	E' → + T E'	E' → + T E'	E' → + T E'
T → E ~ T	E' → ε	E' → ε	E' → ε
T → id	T → E ~ T	T → T E' ~ T	T → id T'
	T → id	T → id	T' → E' ~ T T'
			T' → ε

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - → Context-free grammars ✓
  - → Ambiguity ✓
- Top-down parsers
  - → Algorithm & its problem with left recursion ✓
  - → Left-recursion removal ✓
- Predictive top-down parsing
  - → The LL(1) condition today
  - → Simple recursive descent parsers today
  - → Table-driven LL(1) parsers today

Picking the "Right" Production

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

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

How much lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are LL(1) and LR(1) grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some rhs $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \underline{x}\ \gamma$, for some γ

We will defer the problem of how to compute FIRST sets until we look at the LR(1) table construction algorithm

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some rhs $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct
See the next slide

Predictive Parsing

What about ε-productions?

⇒ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(\alpha)$, too

Define $\text{FIRST}^{+}(\alpha)$ as

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(\alpha)$, if $\varepsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$, otherwise

Then, a grammar is LL(1) iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^{+}(\alpha) \cap \text{FIRST}^{+}(\beta) = \varnothing$$

FOLLOW(α) is the set of all words in the grammar that can legally appear immediately after an α

Predictive Parsing

Given a grammar that has the LL(1) property

- Can write a simple routine to recognize each lhs
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$$

```
/* find an A */
if (current_word ∈ FIRST(β1))
    find a β1 and return true
else if (current_word ∈ FIRST(β2))
    find a β2 and return true
else if (current_word ∈ FIRST(β3))
    find a β3 and return true
else
    report an error and return false
```

Grammars with the LL(1) property are called predictive grammars because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the LL(1) property are called predictive parsers.

One kind of predictive parser is the recursive descent parser.

Of course, there is more detail to "find a β_i" (§ 3.3.4 in EAC)

Recursive Descent Parsing

Recall the expression grammar, after transformation

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	number
11		|	id

This produces a parser with six mutually recursive routines:

- Goal
- Expr
- EPrime
- Term
- TPrime
- Factor

Each recognizes one NT or T

The term descent refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
       then next compilation step;
       else
          report syntax error;
          return false;

Expr( )
   if (Term( ) = false)
     then return false;
     else return Eprime( );
```

looking for EOF, found token

```
Factor( )
   if (token = Number) then
      token ← next_token( );
      return true;
   else if (token = Identifier) then
       token ← next_token( );
       return true;
   else
      report syntax error;
      return false;
```

EPrime, *Term*, & *TPrime* follow the same basic lines (Figure 3.7, EAC)

looking for Number or Identifier, found token instead

Recursive Descent Parsing

To build a parse tree:

- Augment parsing routines to build nodes
- Pass nodes between routines using a stack
- Node for each symbol on rhs
- Action is to pop rhs nodes, make them children of lhs node, and push this subtree

To build an abstract syntax tree

- Build fewer nodes
- Put them together in a different order

```
Expr( )
    result ← true;
    if (Term( ) = false)
      then return false;
      else if (EPrime( ) = false)
              then result ← false;
              else
                build an Expr node
                pop EPrime node
                  pop Term node
                make EPrime & Term
                    children of Expr
                push Expr node
    return result;
```

Success ⇒ build a piece of the parse tree

Left Factoring

What if my grammar does not have the LL(1) property?

⇒ Sometimes, we can transform the grammar

The Algorithm

∀ $A \in NT$,
  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,
    $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$,

  with
    $A \rightarrow \alpha Z \mid \gamma$
    $Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

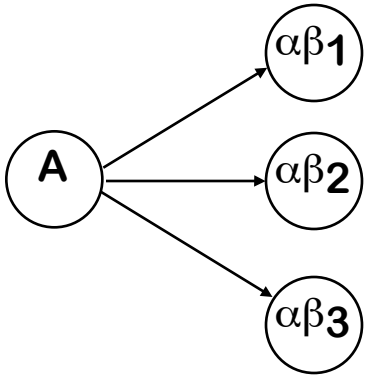
  where Z is a new element of NT

Repeat until no common prefixes remain

Left Factoring

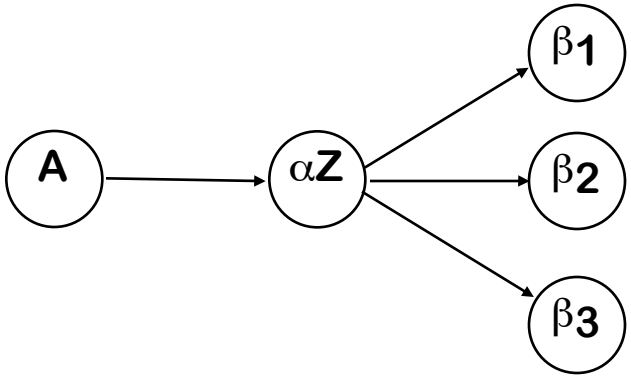
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$



becomes …

$A \rightarrow \alpha Z$
$Z \rightarrow \beta_1 \mid \beta_2 \mid \beta_n$



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*
Arguments	→	[*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