Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Top-Down Parsing

Parsing Techniques

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

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), shift-reduce parsers)*

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step, a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars

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 production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
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
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

The key is picking the right production in step 1

- That choice should be guided by the input string

Example

Using version with correct precedence and associativity

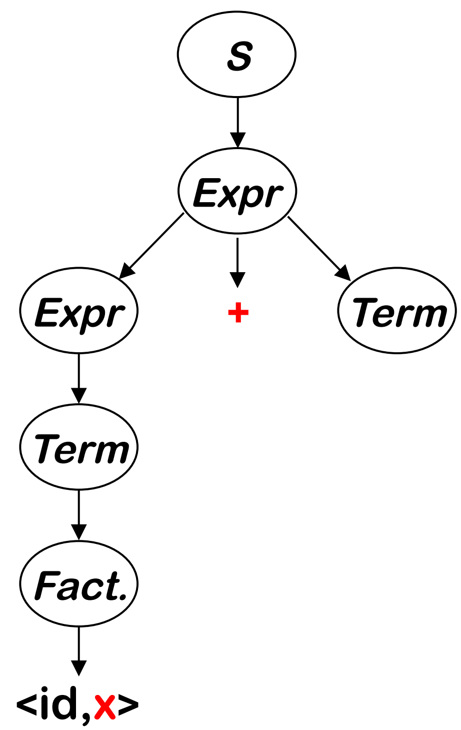
1	*S*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	num
9		|	id

And the input: x – 2 * y

Example

Let's try x – 2 * y :

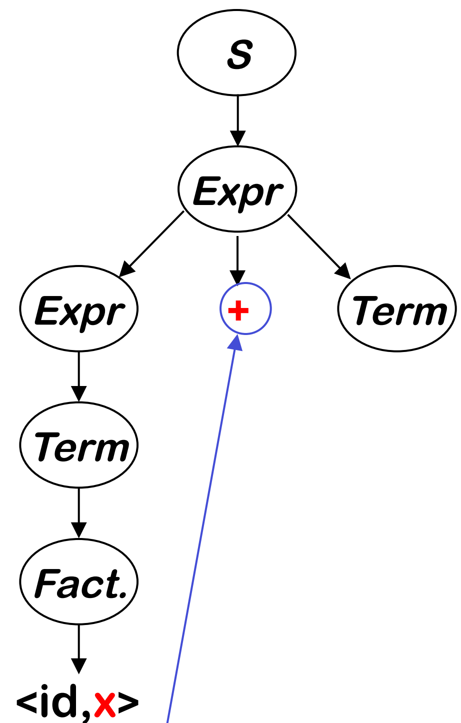
Rule	Sentential Form	Input
-	*S*	↑ x – 2 * y
1	*Expr*	↑ x – 2 * y
2	*Expr* + *Term*	↑ x – 2 * y
4	*Term* + *Term*	↑ x – 2 * y
7	*Factor* + *Term*	↑ x – 2 * y
9	<id,x> + *Term*	↑ x – 2 * y
	<id,x> + *Term*	x ↑ – 2 * y



Example

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



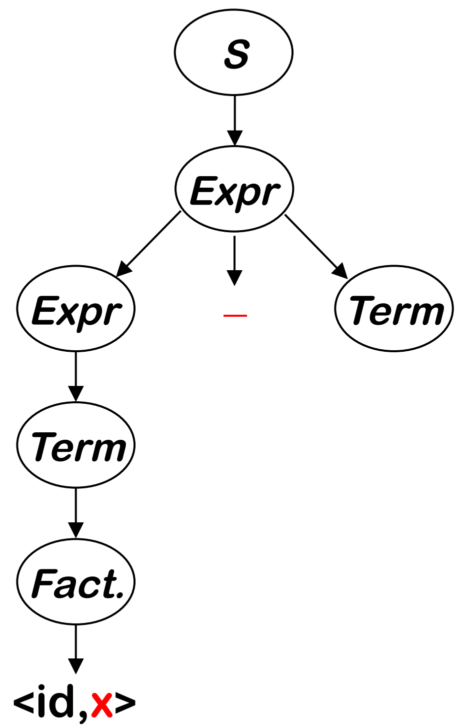
Note that "–" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x – 2 * y :

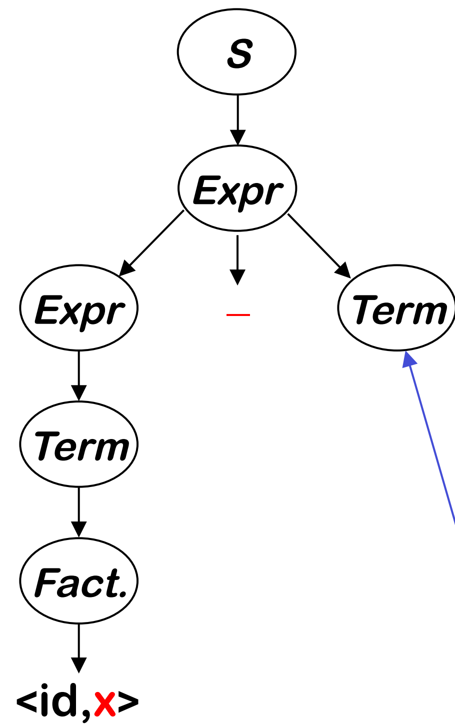
Rule	Sentential Form	Input
-	*S*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr – Term*	↑x – 2 * y
4	*Term – Term*	↑x – 2 * y
7	*Factor – Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
-	<id,x> – *Term*	x ↑– 2 * y
-	<id,x> – *Term*	x – ↑2 * y



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

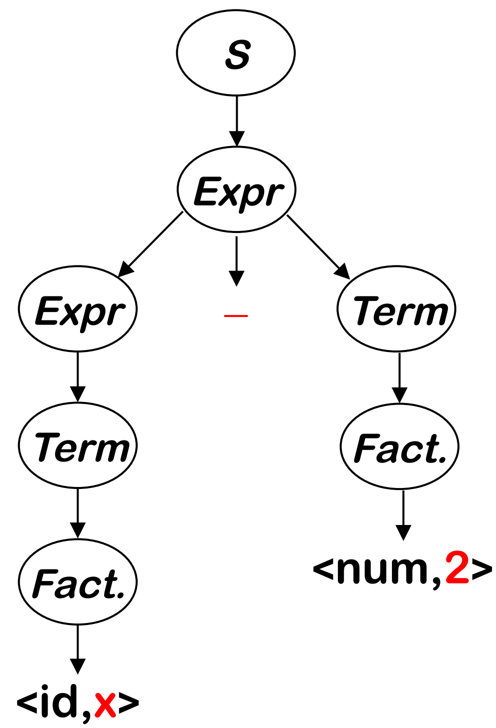
We can advance past "–" to look at "2"

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

Example

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?

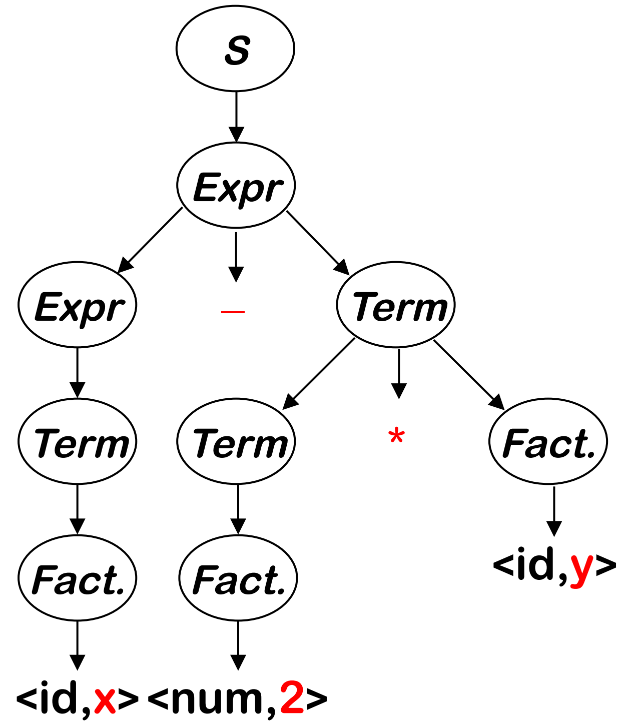
- num matches "2"
- We have more input, but no *NT*s left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

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 parse

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

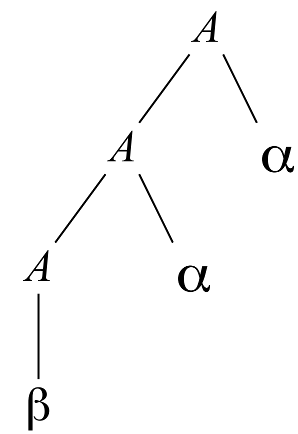
Formally,

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 $\alpha \in (NT \cup T)^{+}$

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)

Eliminating Immediate Left Recursion



To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

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

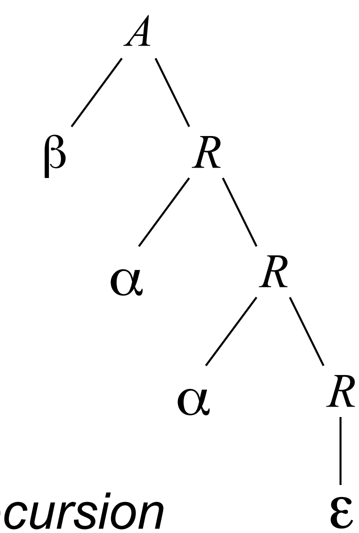
where α or β are strings of terminal and non-terminal symbols and neither α nor β start with *A*

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



Eliminating Immediate 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

Eliminating Immediate Left Recursion

Substituting back into the grammar yields

1	*S*	→	*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*	→	num
11		|	id

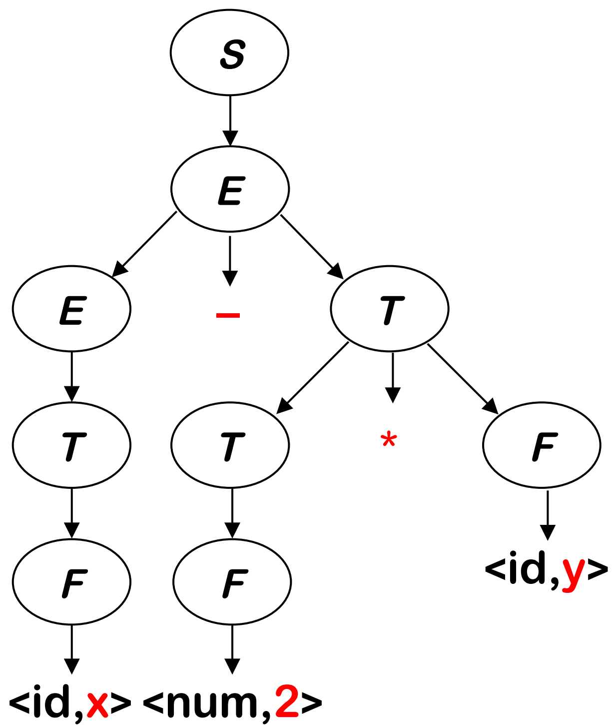
- **This grammar is correct, if somewhat non-intuitive.**
- **A top-down parser will terminate using it.**

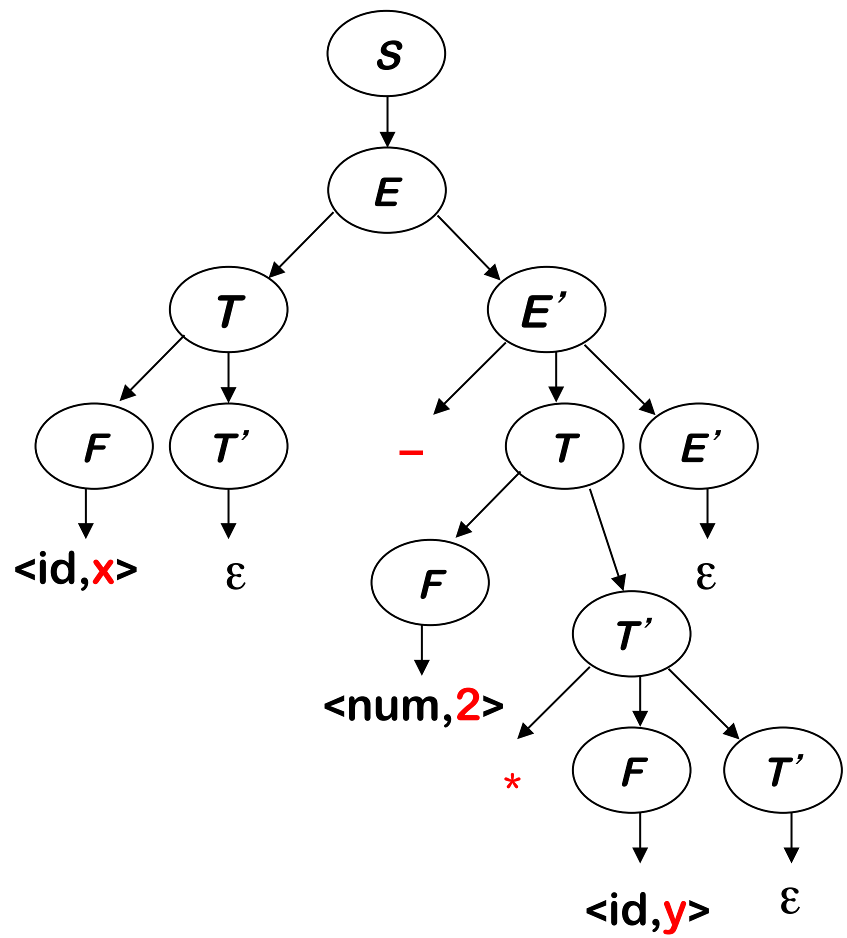
Left-Recursive and Right-Recursive Grammar

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

1	*S*	→	*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*	→	num
11		|	id

Preserves Precedence





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* $i\text{-}1$

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

$A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$, *where* $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
are all the current productions for A_j

eliminate any immediate left recursion on A_i *using the direct transformation*

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$),
and no epsilon productions ($A_i \rightarrow \varepsilon$)

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_j in its *rhs,* for $j < i$
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 and 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 $s < k$

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 look-ahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami, or Earley's algorithm
  - Complexity is $O(|x|^3)$ where *x* is the input string

Fortunately,

- Large subclasses of context free grammars can be parsed efficiently with limited look-ahead
  - Linear complexity, $O(|x|)$ where x is the input string
- Most programming language constructs fall in those subclasses

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

Left-Recursive and Right-Recursive Grammar

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

1	*S*	→	*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*	→	num
11		|	id

Why is this better? It is no longer left recursive so we eventually need to eat a token before we can continue expanding our parse tree…. We can use this token to help us figure out which rule to apply.

Enter Predictive Parsing

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β based on peeking at the next token in the stream

FIRST sets

For a string of grammar symbols α, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $x \in$ FIRST(α) *iff* $\alpha \Rightarrow^* x\, \gamma$, for some γ

($\Rightarrow^*$ means a bunch of (0 or more) productions applied in series)

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 look-ahead of exactly one symbol !

(Pursuing this idea leads to LL(1) parser generators...)

Recursive Descent Parsing

Recursive-descent parsing

- A top-down parsing method
- The term *descent* refers to the direction in which the parse tree is traversed (or built).
- Use a set of *mutually recursive* procedures (one procedure for each non-terminal symbol)
  - Start the parsing process by calling the procedure that corresponds to the start symbol
  - Each production becomes one clause in procedure
- We consider a special type of recursive-descent parsing called predictive parsing
  - *Use a look-ahead symbol to decide which production to use*

Recursive Descent Parsing

1	*S*	→	if *E* then *S* else *S*
2		|	begin *S L*
3		|	print *E*
4	*L*	→	end
5		|	; *S L*
6	*E*	→	num = num

```
void S() {
  switch(lookahead) {
    case IF: match(IF); E(); match(THEN); S();
             match(ELSE); S(); break;
    case BEGIN: match(BEGIN); S(); L(); break;
    case PRINT: match(PRINT); E(); break;
    default: error();
  }
}

void E() { match(NUM); match(EQ); match(NUM); }
```

```
void match(int token) {
        if (lookahead==token)
                lookahead=getNextToken();
        else
                error();
}
```

```
void L() {
  switch(lookahead) {
    case END: match(END); break;
    case SEMI: match(SEMI); S();
               L(); break;
    default: error();
  }
}

void main() {
  lookahead=getNextToken();
  S();
  match(EOF);
}
```

Execution For Input: if 2=2 then print 5=5 else print 1=1

main: call S();

- S_1: find the production for (S, IF) : $S \rightarrow$ if E then S else S
- S_1: match(IF);
- S_1: call E();
    - E_1: find the production for (E, NUM): $E \rightarrow$ num = num
    - E_1: match(NUM); match(EQ); match(NUM);
    - E_1: return from E_1 to S_1
- S_1: match(THEN);
- S_1:call S();
    - S_2: find the production for (S, PRINT): $S \rightarrow$ print E
    - S_2: match(PRINT);
    - S_2: call E();
        - E_2: find the production for (E, NUM): $E \rightarrow$ num = num
        - E_2: match(NUM); match(EQ); match(NUM);
        - E_2: return from E_2 to S_2
- S_2: return from S_2 to S_1
- S_1: match(ELSE);
- S_1: call S();
    - S_3: find the production for (S, PRINT): $S \rightarrow$ print E
    - S_3: match(PRINT);
    - S_3: call E();
        - E_3: find the production for (E, NUM): $E \rightarrow$ num = num
        - E_3: match(NUM); match(EQ); match(NUM);
        - E_3: return from E_2 to S_3
- S_3: return from S_3 to S_1
- S_1: return from S_1 to main

main: match(EOF); return success;

Left Factoring

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

- 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

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

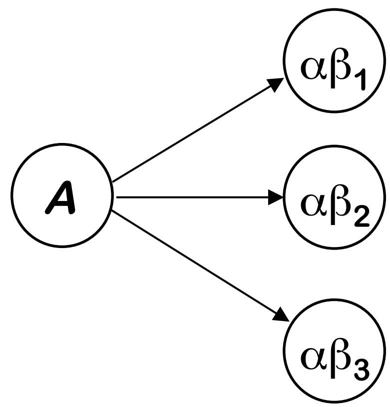
where Z is a new element of NT

Repeat until no common prefixes remain

Left Factoring

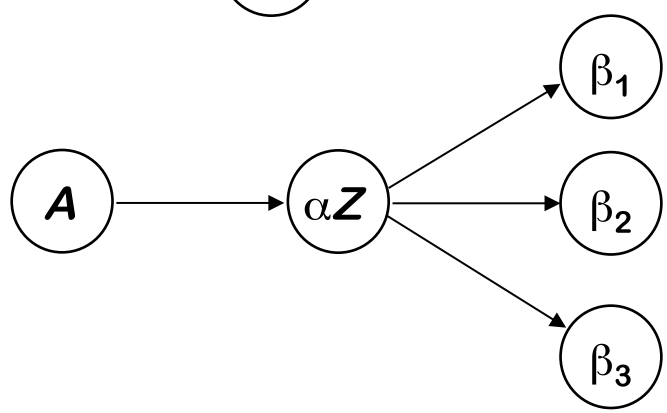
A graphical explanation for the left-factoring

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



becomes …

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



Left Factoring - Example

Consider the following fragment of the expression grammar

1	***Factor*** →	**Id**	
2	|	**Id** [***ExprList***]	
3	|	**Id** (***ExprList***)	

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

After left factoring, it becomes

1	***Factor***	→	**Id *Arguments***
2	***Arguments***	→	**[*ExprList*]**
3		|	**(*ExprList*)**
4		|	ε

This grammar accepts the same language, and it has the the *LL(1)* property

FIRST(rhs_1) = { Id }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = ?
(Intuitively, we can think of the FOLLOW of *Arguments* as the first of rhs_4)
FOLLOW(*Arguments*)=FOLLOW(*Factor*) = { $ }
They are all distinct
⇒ Grammar has the *LL(1)* property