Parsing — Part II

(Top-down parsing, left-recursion removal)

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" $\Rightarrow$ 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

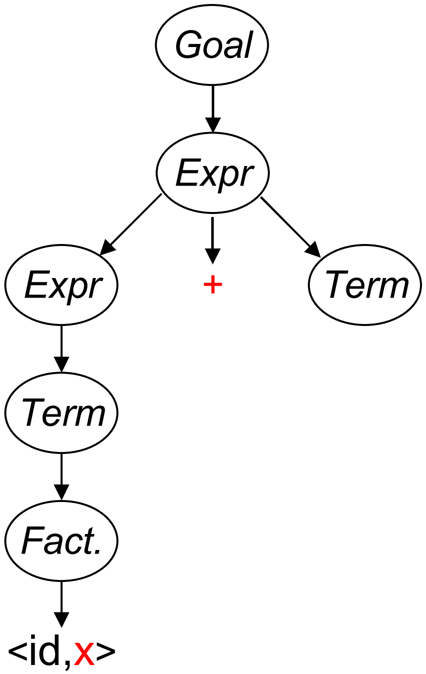
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

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

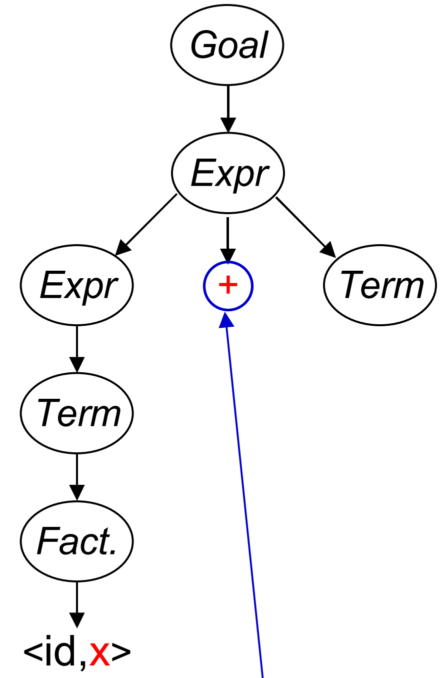


Leftmost derivation, choose productions in an order that exposes problems

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



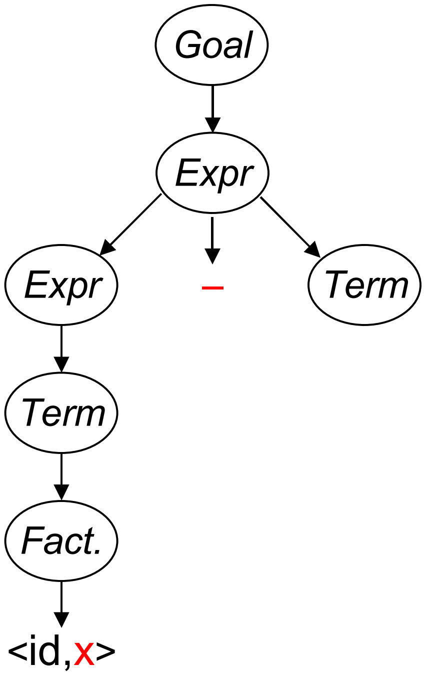
This worked well, except that "–" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x - 2 * y :

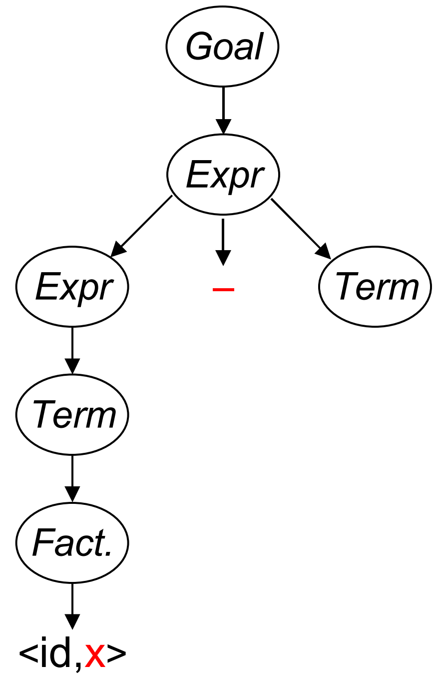
Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr* – *Term*	↑x – 2 * y
4	*Term* – *Term*	↑x – 2 * y
7	*Factor* – *Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



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

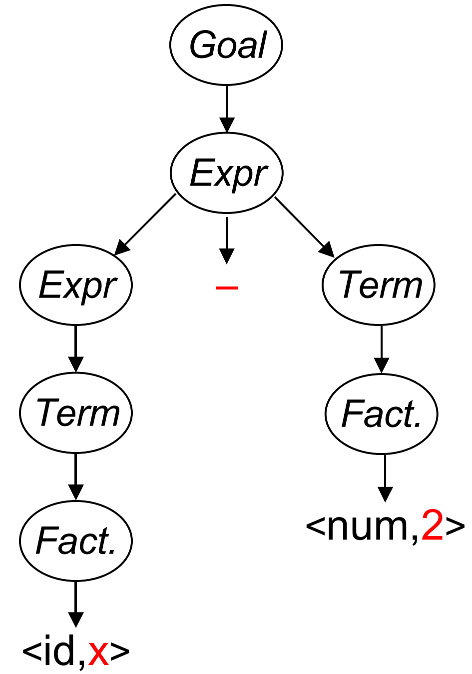
We can advance past "–" to look at "2"

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

Example

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

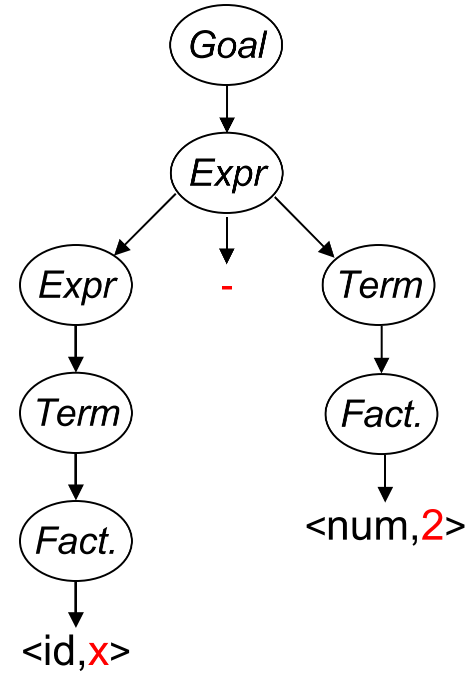
Rule	*Sentential Form*	*Input*
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2↑ * y



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?

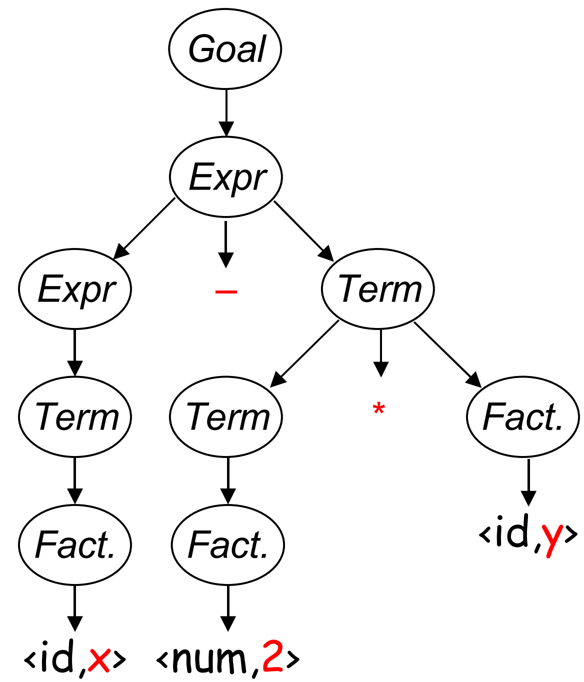
- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ 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

$$\begin{aligned} Fee &\rightarrow Fee\ \alpha \\ &\quad \mid\ \beta \end{aligned}$$

where neither α nor β start with *Fee*

We can rewrite this as

$$\begin{aligned} Fee &\rightarrow \beta\ Fie \\ Fie &\rightarrow \alpha\ Fie \\ &\quad \mid\ \varepsilon \end{aligned}$$

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 $i \leftarrow 1$ *to* n

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

for $s \leftarrow 1$ *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

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*

$G \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow E \sim T$

$T \rightarrow \underline{\text{id}}$

Example

- Order of symbols: *G, E, T*

1. $A_i = G$

$G \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow E \sim T$

$T \rightarrow$ <u>id</u>

Example

- Order of symbols: *G, E, T*

1. $A_i = G$	2. $A_i = E$
$G \rightarrow E$	$G \rightarrow E$
$E \rightarrow E + T$	$E \rightarrow T\ E'$
$E \rightarrow T$	$E' \rightarrow +\ T\ E'$
$T \rightarrow E \sim T$	$E' \rightarrow \varepsilon$
$T \rightarrow \underline{\text{id}}$	$T \rightarrow E \sim T$
	$T \rightarrow \underline{\text{id}}$

Example

- Order of symbols: G, E, T

1. $A_i = G$	2. $A_i = E$	3. $A_i = T$, $A_s = E$
$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$
$E \rightarrow E + T$	$E \rightarrow T E'$	$E \rightarrow T E'$
$E \rightarrow T$	$E' \rightarrow + T E'$	$E' \rightarrow + T E'$
$T \rightarrow E \sim T$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
$T \rightarrow$ <u>id</u>	$T \rightarrow E \sim T$	$T \rightarrow T E' \sim T$
	$T \rightarrow$ <u>id</u>	$T \rightarrow$ <u>id</u>

Go to Algorithm

Example

- Order of symbols: *G, E, T*

1. $A_i = G$

$G \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow E \sim T$

$T \rightarrow \text{id}$

2. $A_i = E$

$G \rightarrow E$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow \varepsilon$

$T \rightarrow E \sim T$

$T \rightarrow \text{id}$

3. $A_i = T,\ A_s = E$

$G \rightarrow E$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow \varepsilon$

$T \rightarrow T\ E' \sim T$

$T \rightarrow \text{id}$

4. $A_i = T$

$G \rightarrow E$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow \varepsilon$

$T \rightarrow \text{id}\ T'$

$T' \rightarrow E' \sim T\ T'$

$T' \rightarrow \varepsilon$