Top Down Parsing - Part I

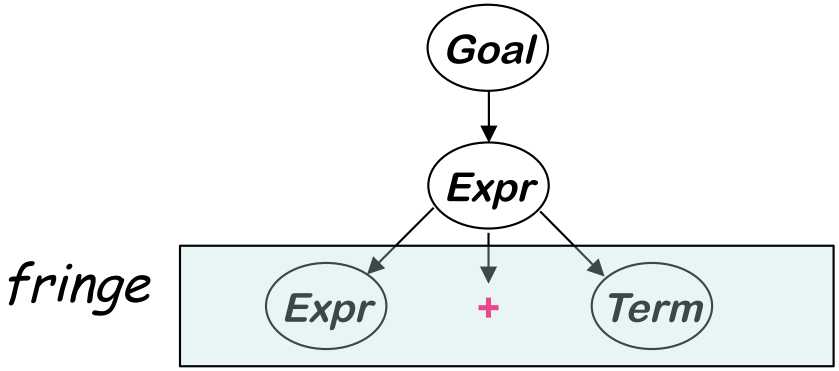
Parsing Techniques

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

- Start at root of 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

Top-down Parsing

- *Starts with root of parse tree*
- *Root node is labeled with goal symbol*
- *Expand all non-terminals (NT) at fringe of tree*



Top-down parsing algorithm

Construct the root node of parse tree

Repeat until lower fringe matches input string

1. *At node labeled A, select production with A on LHS and, for each symbol on RHS, construct appropriate child*
2. *If terminal symbol added to fringe doesn't match input, backtrack*
3. *Find the next node (NT) to be expanded*

The key is picking the right production in step 1

- *That choice should be guided by the input string*

Remember the expression grammar?

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

And the input x – 2 * y

Example

Let's try x – 2 * y :

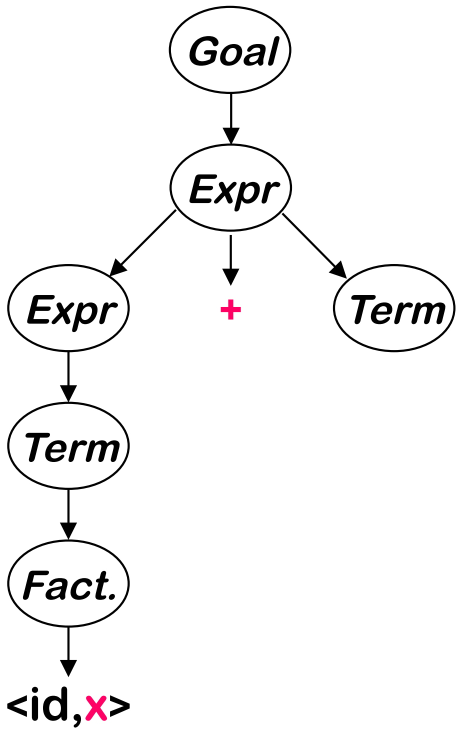
Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y

↑ *is the position in the input buffer*

Example

Let's try x – 2 * y :

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr* + *Term*	↑x - 2 * y
3	*Term* + *Term*	↑x - 2 * y
6	*Factor* + *Term*	↑x - 2 * y
9	<*id*,x> + *Term*	↑x - 2 * y
→	<*id*,x> + *Term*	x ↑- 2 * y



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

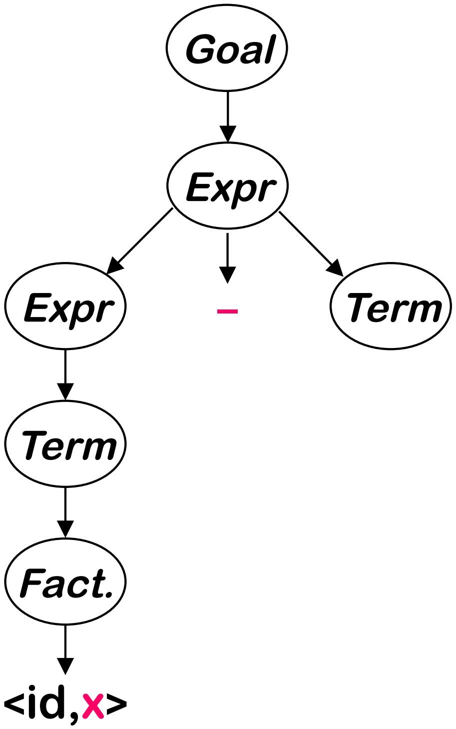
The parser must backtrack to here

↑ *is the position in the input buffer*

Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
2	*Expr - Term*	↑x - 2 * y
3	*Term - Term*	↑x - 2 * y
6	*Factor - Term*	↑x - 2 * y
9	<id,x> - *Term*	↑x - 2 * y
→	<id,x> - *Term*	x ↑- 2 * y
→	<id,x> - *Term*	x - ↑2 * y



Now, "-" and "-" match

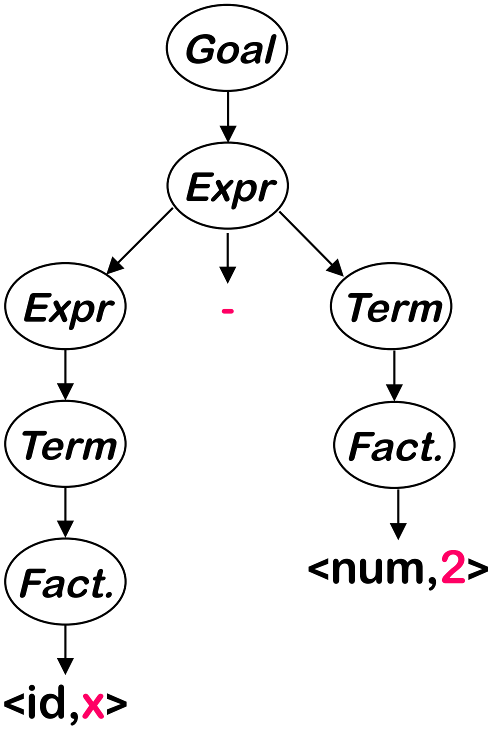
Now we can expand Term to match "2"

⇒ 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
6	<id,x> - Factor	x - ↑2 * y
8	<id,x> - <num,2>	x - ↑2 * y
→	<id,x> - <num,2>	x - 2 ↑* y



Where are we?

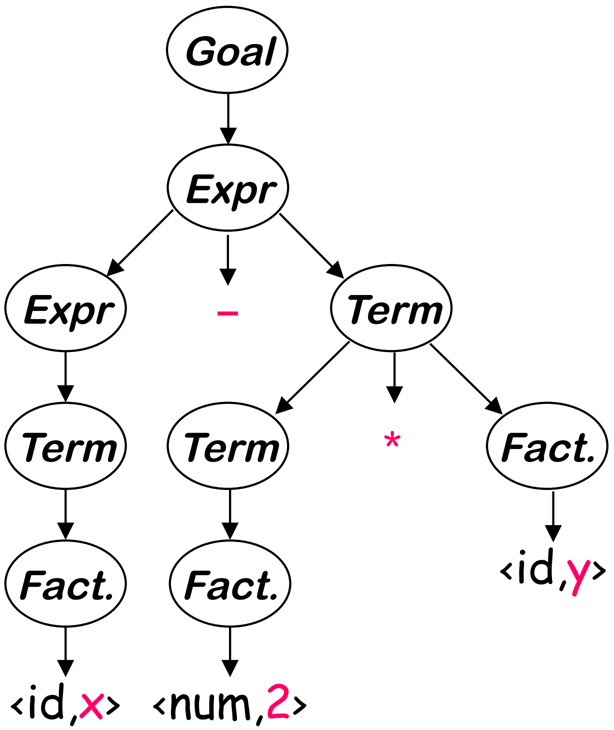
- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

Trying again with "2" in x – 2 * y :

Rule	Sentential Form	Input
→	‹id,x› - *Term*	x - ↑2 * y
4	‹id,x› - *Term* * *Factor*	x - ↑2 * y
6	‹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↑



The Point:

The parser must make the right choice when it expands a NT.
Wrong choices lead to wasted effort.

Another possible parse

Other choices for expansion are possible

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr* + *Term*	↑x - 2 * y
1	*Expr* + *Term* + *Term*	↑x - 2 * y
1	*Expr* + *Term* + *Term* + *Term*	↑x - 2 * y
1	*And so on ....*	↑x - 2 * y

Consumes no input!

This expansion doesn't terminate

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

Left Recursion

Our classic expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- In top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is always a bad property in a compiler

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

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

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

where neither α nor β start with *Fee*

We can rewrite this fragment as

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

where *Fie* is a **new non-terminal**

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

$$\begin{array}{lcl} Expr & \rightarrow & Expr + Term \\ & \mid & Expr - Term \\ & \mid & Term \end{array} \qquad \begin{array}{lcl} Term & \rightarrow & Term * Factor \\ & \mid & Term * Factor \\ & \mid & Factor \end{array}$$

$$\begin{array}{lcl} Fee & \rightarrow & Fee \ \alpha \\ & \mid & \beta \end{array}$$

$$\begin{array}{lcl} Fee & \rightarrow & \beta \ Fie \\ Fie & \rightarrow & \alpha \ Fie \\ & \mid & \varepsilon \end{array}$$

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr → *Expr* + *Term*
| *Expr* - *Term*
| *Term*

Term → *Term* * *Factor*
| *Term* * *Factor*
| *Factor*

Applying the transformation yields

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

Term → *Factor* *Term′*
Term′ → * *Factor* *Term′*
| / *Factor* *Term′*
| ε

These fragments use only right recursion

Eliminating Left Recursion

Substituting them back into the grammar yields

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

- This grammar is correct, if somewhat non-intuitive.
- 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 $A_1, A_2, …, A_n$
for $i \leftarrow 1$ to $n$
    for $s \leftarrow 1$ to $i - 1$
        replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid … \mid \delta_k\gamma$,
            where $A_s \rightarrow \delta_1 \mid \delta_2 \mid … \mid \delta_k$ 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

1. $A_i = G$

$G \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow E * T$

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

2. $A_i = E$

$G \rightarrow E$

$E \rightarrow T E'$

$E' \rightarrow + T E'$

$E' \rightarrow \varepsilon$

$T \rightarrow E * T$

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

3. $A_i = T, A_s = E$

$G \rightarrow E$

$E \rightarrow T E'$

$E' \rightarrow + T E'$

$E' \rightarrow \varepsilon$

$T \rightarrow T E' * T$

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

4. $A_i = T$

$G \rightarrow E$

$E \rightarrow T E'$

$E' \rightarrow + T E'$

$E' \rightarrow \varepsilon$

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

$T' \rightarrow E' * T T'$

$T' \rightarrow \varepsilon$

Go to Algorithm