Parsing — Part II

(Ambiguity, Top-down parsing, Left-recursion Removal)

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single *sentential form*, the grammar is ambiguous
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the *if*-*then*-*else* problem

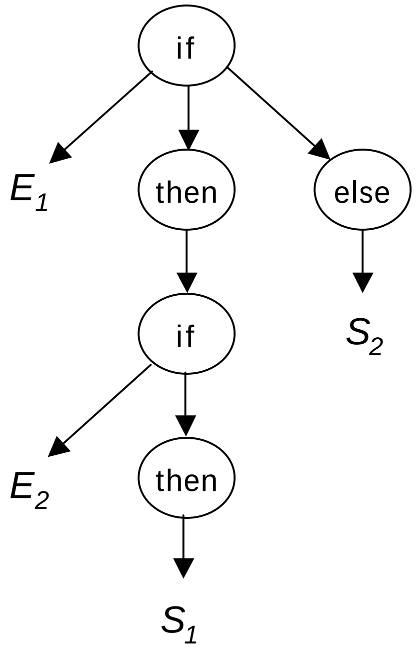
Stmt ® if *Expr* then *Stmt*
  | if *Expr* then *Stmt* else *Stmt*
  | *… other stmts …*

This ambiguity is entirely grammatical in nature

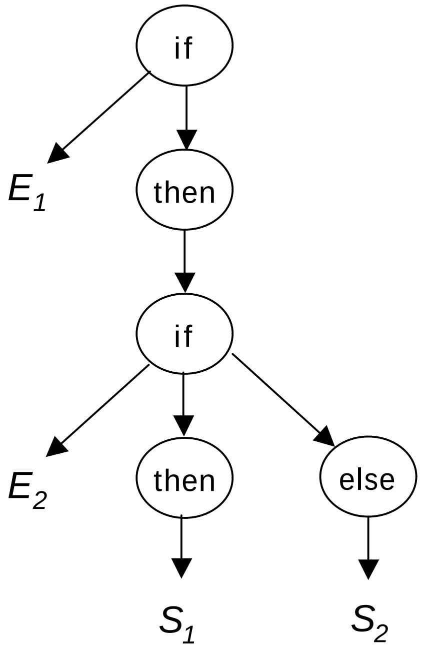
Ambiguity

This sentential form has two derivations (If a derivation has more than 1 parse tree, the grammar is ambiguous

$\underline{\text{if}}\ \mathit{Expr}_1\ \underline{\text{then}}\ \underline{\text{if}}\ \mathit{Expr}_2\ \underline{\text{then}}\ \mathit{Stmt}_1\ \ \underline{\text{else}}\ \mathit{Stmt}_2$



production 2, then production 1



production 1, then production 2

Ambiguity

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if (common sense rule)

1	*Stmt*	→	*WithElse*
2		|	*NoElse*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	… other stmts …
5	*NoElse*	→	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

With this grammar, the example has only one derivation

Ambiguity

if *Expr*$_1$ then if *Expr*$_2$ then *Stmt*$_1$ else *Stmt*$_2$

Rule	*Sentential Form*
—	*Stmt*
2	*NoElse*
5	if *Expr* then *Stmt*
?	if E_1 then *Stmt*
1	if E_1 then *WithElse*
3	if E_1 then if *Expr* then *WithElse* else *WithElse*
?	if E_1 then if E_2 then *WithElse* else *WithElse*
4	if E_1 then if E_2 then S_1 else *WithElse*
4	if E_1 then if E_2 then S_1 else S_2

This binds the else controlling S_2 to the inner if

Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

a = f(17)

In some languages, f could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of *type*, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must to handle these with a different mechanism
  - Step outside grammar rather than use a more complex grammar

Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  - Knowledge of declarations, types, ...
  - Accept a superset of *L(G)* & check it with other means[†]
  - This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- Parsing techniques that “do the right thing”

[†]See Chapter 4

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 Î 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

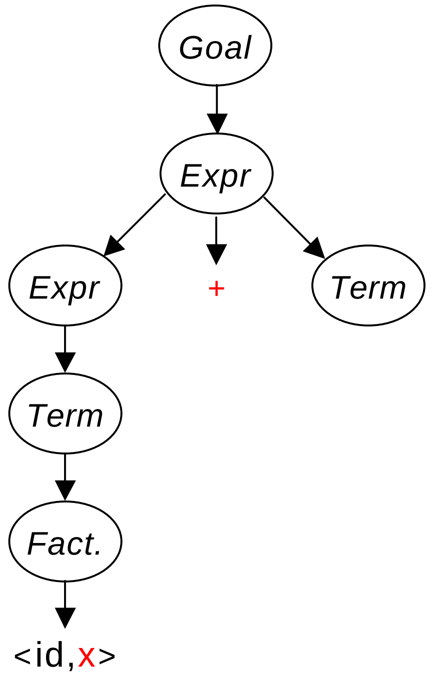
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

And the input x - 2 * y

Example

Let’s try x - 2 * y :

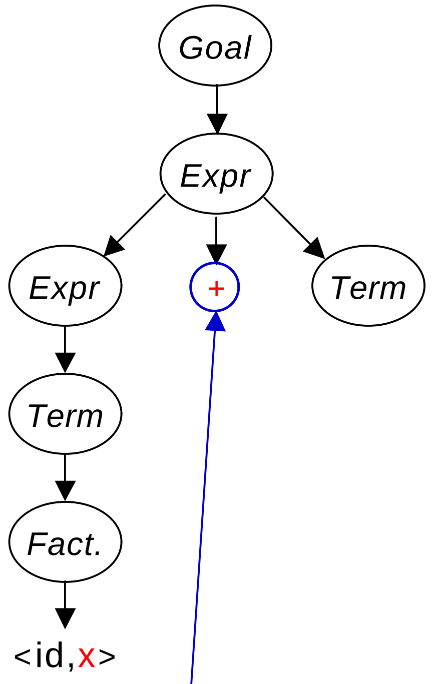
Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id, x> + *Term*	↑x - 2 * y
9	<id, x> + *Term*	x ↑- 2 * y



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



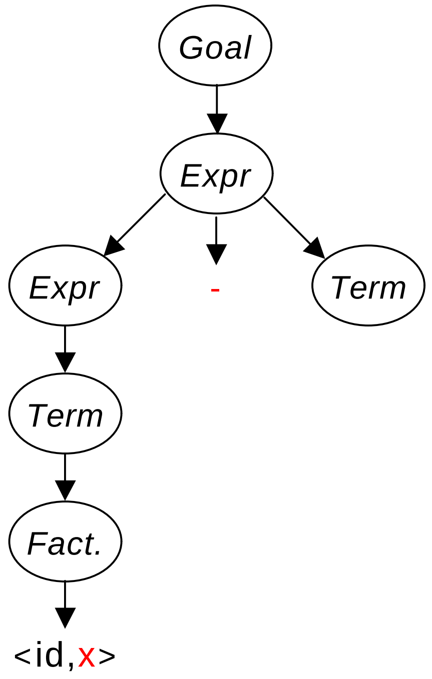
This worked well, except that "-" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x - 2 * y :

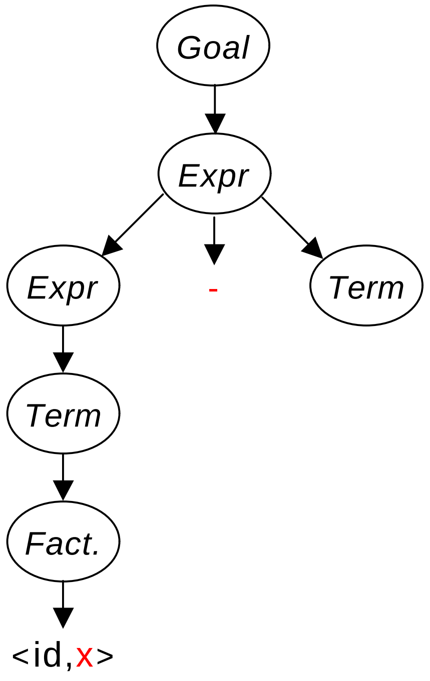
Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
3	*Expr* - *Term*	↑x - 2 * y
4	*Term* - *Term*	↑x - 2 * y
7	*Factor* - *Term*	↑x - 2 * y
9	<id, x> - *Term*	↑x - 2 * y
9	<id, x> - *Term*	x ↑- 2 * y
—	<id, x> - *Term*	x - ↑2 * y



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

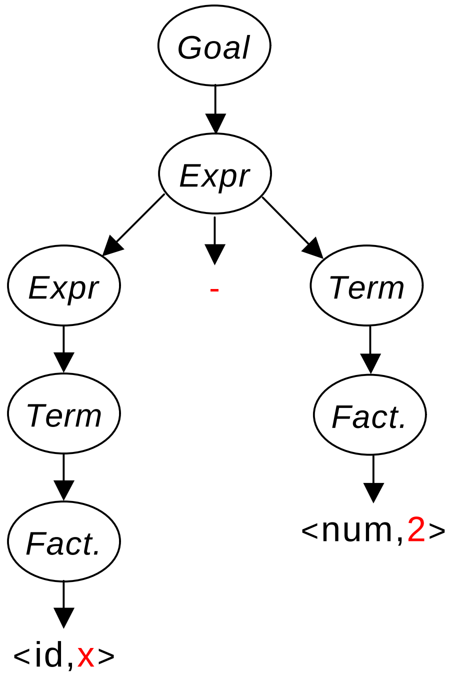
We can advance past "-" to look at "2"

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

Example

Trying to match the “2” in <u>x</u> - <u>2</u> * <u>y</u> :

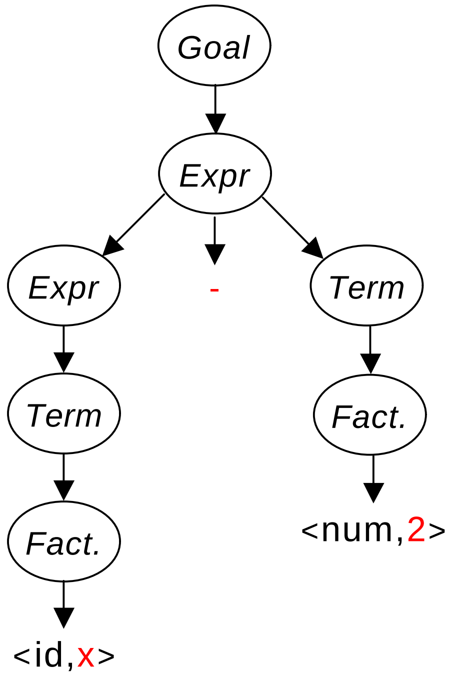
Rule	*Sentential Form*	*Input*
—	<id, x> - *Term*	<u>x</u> - ↑<u>2</u> * <u>y</u>
7	<id, x> - *Factor*	<u>x</u> - ↑<u>2</u> * <u>y</u>
9	<id, x> - <num ,2>	<u>x</u> - ↑<u>2</u> * <u>y</u>
—	<id, x> - <num ,2>	<u>x</u> - <u>2</u> ↑* <u>y</u>



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?

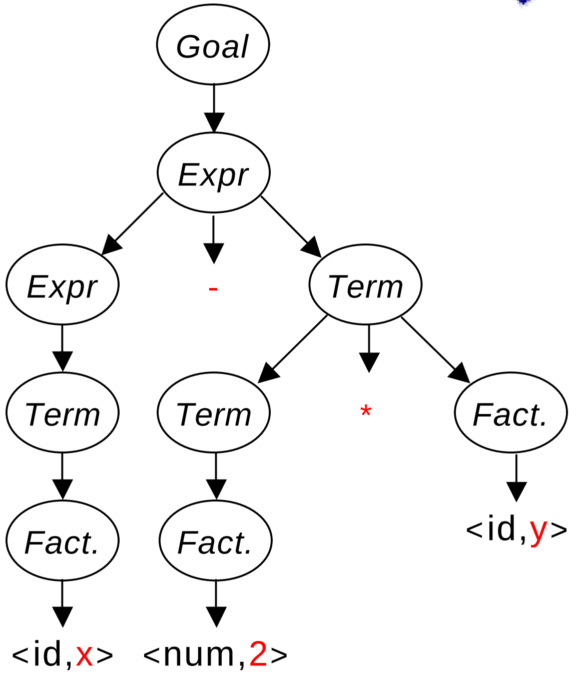
- “2” matches “2”
- We have more input, but no *NT*s left to expand
- The expansion terminated too soon

⇒ 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 & 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

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

where neither α nor β start with *Fee*

We can rewrite this as

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

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

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
Expr & \rightarrow & Expr + Term & Term & \rightarrow & Term * Factor \\
 & | & Expr - Term & & | & Term \ / \ \mathrm{Factor} \\
 & | & Term & & | & Factor
\end{array}
$$

Applying the transformation yields

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
Expr & \rightarrow & Term \ Expr¢ & Term & \rightarrow & Factor \ Term¢ \\
Expr¢ & | & + \ Term \ Expr¢ & Term¢ & | & * \ Factor \ Term¢ \\
 & | & - \ Term \ Expr¢ & & | & / \ Factor \ Term¢ \\
 & | & \varepsilon & & | & \varepsilon
\end{array}
$$

These fragments use only right recursion

They retains the original left associativity

Eliminating Left Recursion

Substituting back into the grammar yields

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

- 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 $A_1$, $A_2$, …, $A_n$
for i ¬ 1 to n
    replace each production $A_i$ ® $A_s$ g with
        $A_i$ ® $d_1$ g ½$d_2$ g½ ¼ ½$d_k$ g,  where $A_s$ ® $d_1$ ½$d_2$½ ¼ ½$d_k$
        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 Þ$^+$ A_i),

and no epsilon productions

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 $s < k$