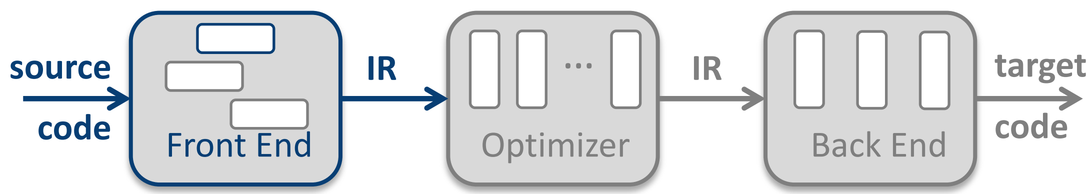
COMP 412
FALL 2018

Parsing II

Top-down parsing

Comp 412



Chapter 3 in EaC2e

Ambiguity

Review

Definitions

- A context-free grammar G is **ambiguous** if there exists has more than one leftmost derivation for some sentence in $L(G)$
- A context-free grammar G is **ambiguous** if there exists has more than one rightmost derivation for some sentence in $L(G)$
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar
  - *However, they must have the same parse tree*

Ambiguity

Review

We talked about syntactic ambiguity

- Syntactic ambiguity in the context-free grammar
  - Classic example is the **if-then-else** grammar

0	*Stmt*	→	<u>if</u> *Expr* <u>then</u> *Stmt*
1		|	<u>if</u> *Expr* <u>then</u> *Stmt* <u>else</u> *Stmt*
2		|	… *other statements* …

  - Fix ambiguity in context-free grammar by re-writing the grammar

- Semantic ambiguity in the underlying language
  - When one set of characters has two possible meanings

    a = f(17,21)
  - Fix ambiguity in meaning by changing the language (e.g., **C**'s [ ]), or
  - Manage ambiguity by accepting language and deferring disambiguation until the compiler has enough context (*e.g.*, type information)

Ambiguity

Semantic ambiguity in the underlying language

One syntax with two meanings

- Classic example arose in Algol-like languages

  a = **f**(17,21)

- Is it a call to a function **f**? or a reference to an element of an array **f**?

Disambiguating this kind of confusion requires context

- Need the value of the declaration for **f**
- The ambiguity is an issue of type, not syntax
- Requires either:
  1. An extra-grammatical solution
     → *Manage the ambiguity by accepting the imprecise language and deferring disambiguation until the compiler has enough context (e.g., type information)*
  2. A different syntax
     → *Fix ambiguity in meaning by changing the language (e.g.,* ***C****'s* [ ] *or BCPL's* !)

Order of Operations or Precedence

Consider again the derivation of x – 2 * y

0	*Expr*	→	*Expr Op Value*
1		|	*Value*
2	*Value*	→	<u>number</u>
3		|	<u>identifier</u>
4	*Op*	→	<u>plus</u>
5		|	<u>minus</u>
6		|	<u>times</u>
7		|	<u>divide</u>

Rule	***Sentential Form***
—	*Expr*
0	*Expr Op Value*
0	*Expr Op Value Op Value*
1	*Value Op Value Op Value*
3	<id,x> *Op Value Op Value*
5	<id,x> – *Value Op Value*
2	<id,x> – <num,2> *Op Value*
6	<id,x> – <num,2> * Value
3	<id,x> – <num,2> * <id,y>

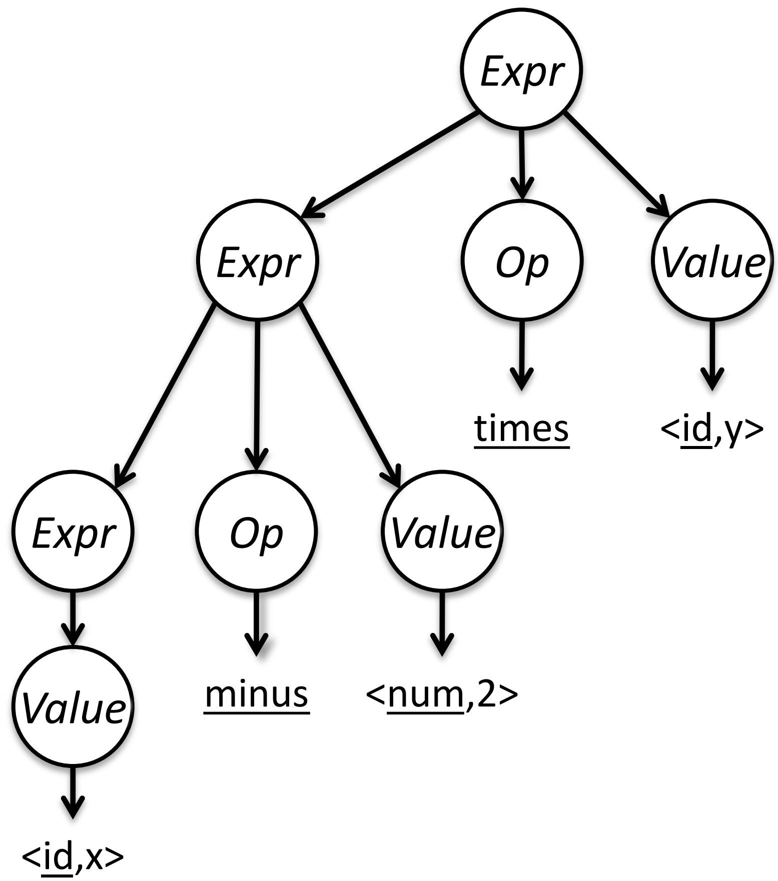
Leftmost Derivation

Not a good expression grammar

Order of Operations

The leftmost derivation is unique, but it specifies the wrong precedence

Rule	*Sentential Form*
—	*Expr*
0	*Expr Op Value*
0	*Expr Op Value Op Value*
1	*Value Op Value Op Value*
3	<id,x> *Op Value Op Value*
5	<id,x> – *Value Op Value*
2	<id,x> – <num,2> *Op Value*
6	<id,x> – <num,2> * Value
3	<id,x> – <num,2> * <id,y>



Evaluates (x - 2) * y

Eliminating ambiguity does not necessarily produce the desired meaning. It produces a consistent meaning, but that meaning can be consistently wrong.

Order of Operations

How do you add precedence to a grammar?

To add precedence

- Decide how many *levels of precedence* the grammar needs
- Create a nonterminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions, including (), +, −, *, and /

- Parentheses first (*level 1*)
- Multiplication and division, next (*level 2*)
- Subtraction and addition, last (*level 3*)

Derivations and Precedence

Adding the standard algebraic precedence produces:

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

The Classic Expression Grammar

The new grammar is larger (7 *vs.* 9)

- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations
- Correctness trumps the speed of the parser

Let's see how it derives x – 2 * y

Both parentheses & precedence are beyond the power of an **RE**

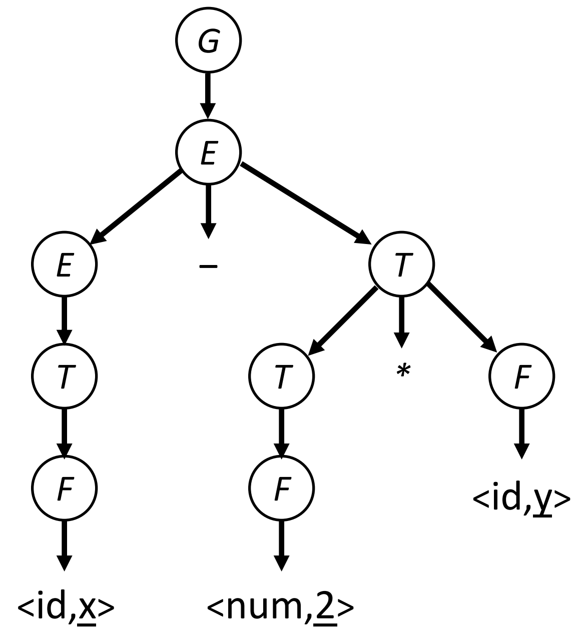
This is the good expression grammar.

Figure 7.7 on p. 351 of EaC2e shows this grammar extended to include Boolean & relational operators.

Derivations and Precedence

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
2	*Expr* – *Term*
3	*Term* – *Term*
6	*Factor* – *Term*
9	<id,x> – *Term*
4	<id,x> – *Term* * *Factor*
6	<id,x> – *Factor* * *Factor*
8	<id,x> – <num,2> * *Factor*
9	<id,x> – <num,2> * <id,y>

The leftmost derivation



Parse tree for x – 2 * y

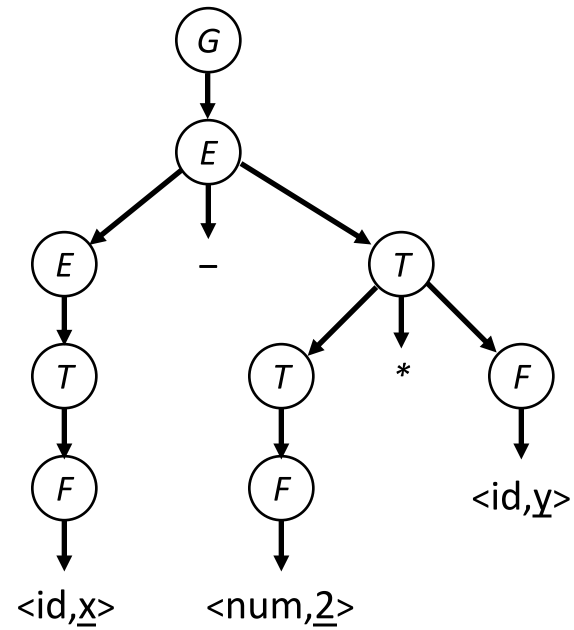
The classic expression grammar derives x – (2 * y) with the parse tree shown..

Both the leftmost and rightmost derivations give the same parse tree and value, because the grammar directly and explicitly encodes the desired precedence.

Derivations and Precedence

Rule	Sentential Form
—	*Goal*
0	*Expr*
2	*Expr* – *Term*
4	*Expr* – *Term* * *Factor*
9	*Expr* – *Term* * <id,y>
6	*Expr* – *Factor* * <id,y>
8	*Expr* – <num,2> * <id,y>
3	*Term* – <num,2> * <id,y>
6	*Factor* – <num,2> * <id,y>
9	<id,x> – <num,2> * <id,y>

The rightmost derivation



Parse tree for x – 2 * y

The classic expression grammar derives x – (2 * y) with the parse tree shown..

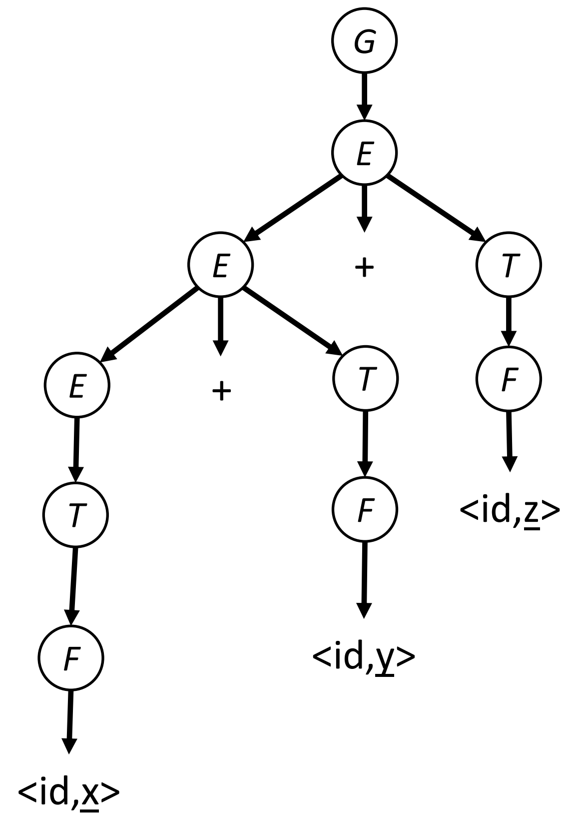
Both the leftmost and rightmost derivations give the same parse tree and value, because the grammar directly and explicitly encodes the desired precedence.

Derivations and Associativity

The classic expression grammar is also left associative

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
1	*Expr* + *Term*
1	*Expr* + *Term* + *Term*
3	*Term* + *Term* + *Term*
6	*Factor* + *Term* + *Term*
9	id + *Term* + *Term*
6	id + *Factor* + *Term*
9	id + id + *Term*
6	id + id + *Factor*
9	id + id + id

Leftmost derivation of "x + y + z"



Parse tree evaluates as (x + y) + z

Corresponds to left-to-right evaluation. We will want to preserve this property.

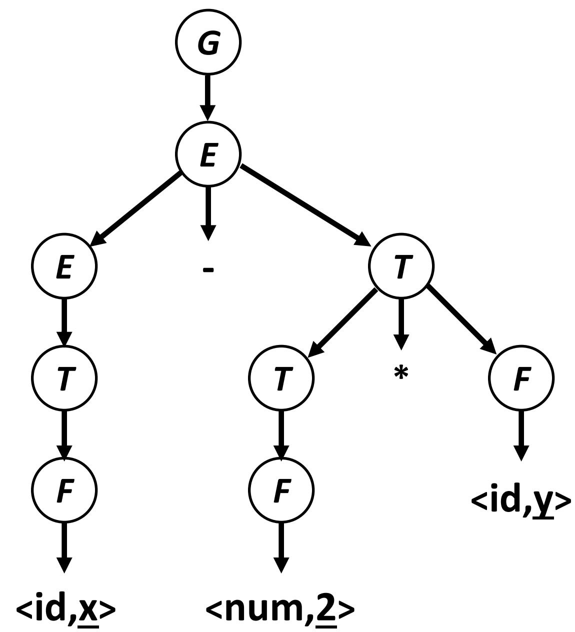
Parsing Techniques

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
- Large class of grammars are backtrack-free

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
- We can make the process deterministic



***Parse tree for* x - 2 * y**

Bottom-up parsers can recognize a larger class of *grammars* than can top-down parsers.

Top-Down Parsing

We will examine two ways of implementing top-down parsers

Recursive-Descent Parser

- Highly efficient, highly flexible form of parser
- Typically implemented as a hand-coded parser
  - Set of mutually-recursive routines
  - Works well for any "backtrack free" or "predictively parsable" grammar
- Easy to understand, easy to implement

Top-Down Parsing

We will examine two ways of implementing top-down parsers

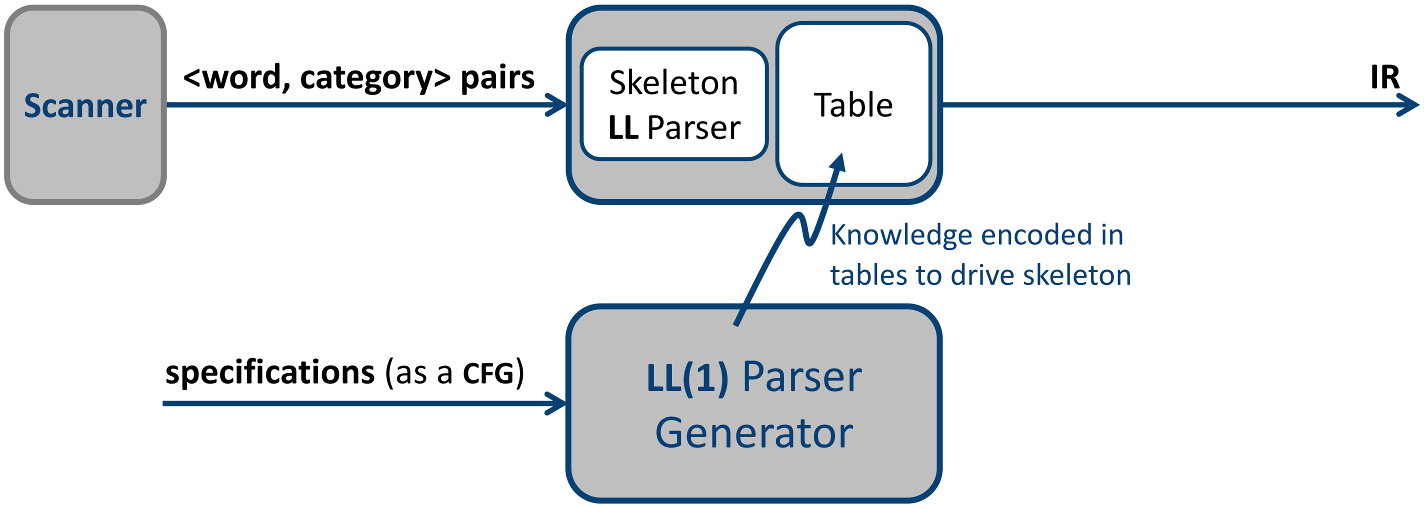
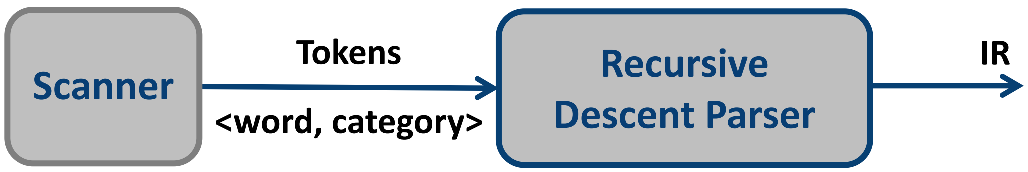


Table-driven LL(1) Parser

- **LL(1)** Parser Generator takes as input a **CFG** that is backtrack free
- Skeleton Parser interprets the table produced by the generator
- In Lab 2, you will implement an **LL(1)** table generator
  - Your table generator will use a recursive-descent parser as its front end

Back to the Meta Question

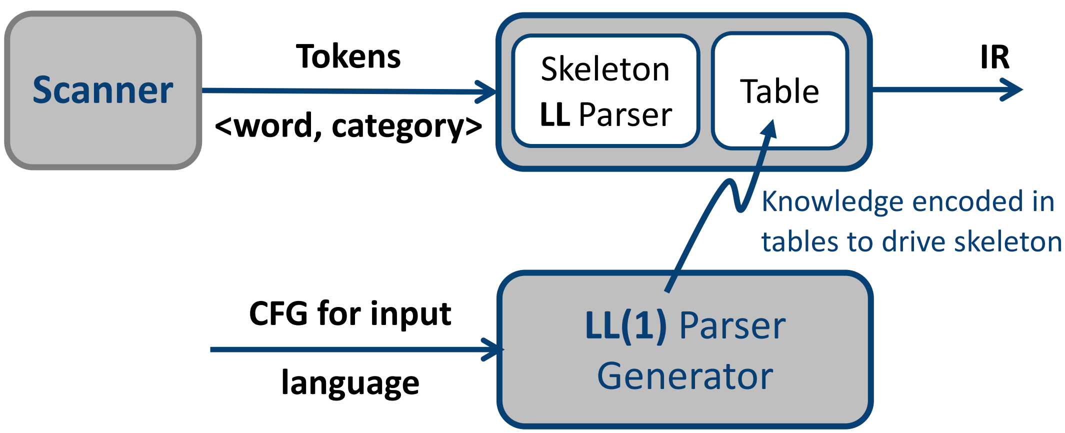
Two kinds of top-down parsers



Design Time
- Write scanner & parser

Build time
- Compile scanner & parser

Compile time
- Run scanner & parser



Design Time
- Write scanner & **CFG**
- Debug **CFG**

Build time
- Invoke parser generator
- Compile scanner & parser

Compile time
- Run scanner & parser

Top-down Parsing

The Algorithm

- A top-down parser starts with the root of the parse tree
- The root node is labeled with the goal symbol of the grammar

Construct the root node of the parse tree

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

1. *At a node labeled with **NT** 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*

The "Classic" Expression Grammar

Consider the Classic 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

Classic Expression Grammar

and the input x – 2 * y

Example

I am going to make choices for pedagogical reasons, rather than for consistency

Let's try to derive x – 2 * y :

Goal

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

Lower fringe of the partially completed parse tree

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		|	**identifier**

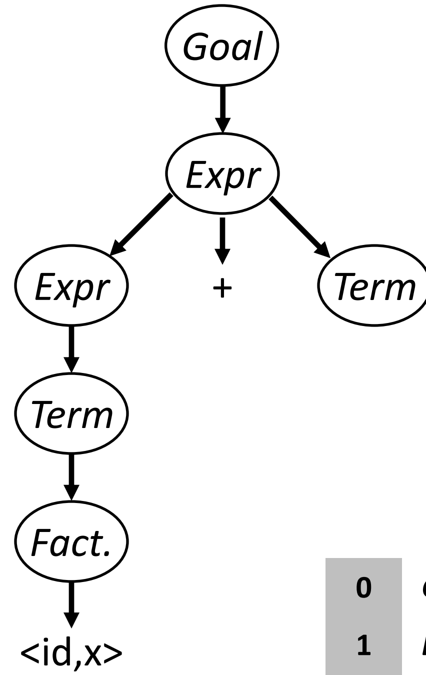
↑ is the position in the input buffer

*Build a **leftmost** derivation, to work with a left-to-right scanner.*

Example

Let's try to derive 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

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		|	identifier

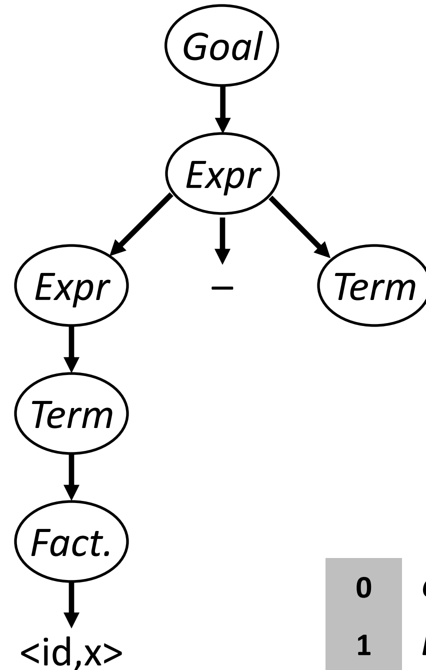
Example

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"



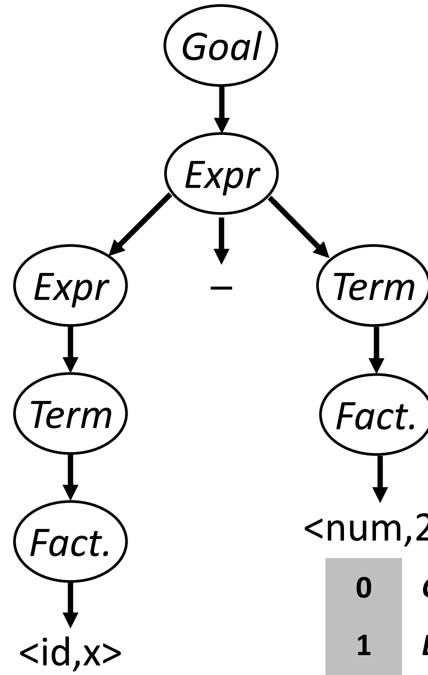
0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr – Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	identifier

⇒ 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



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		|	identifier

Where are we?

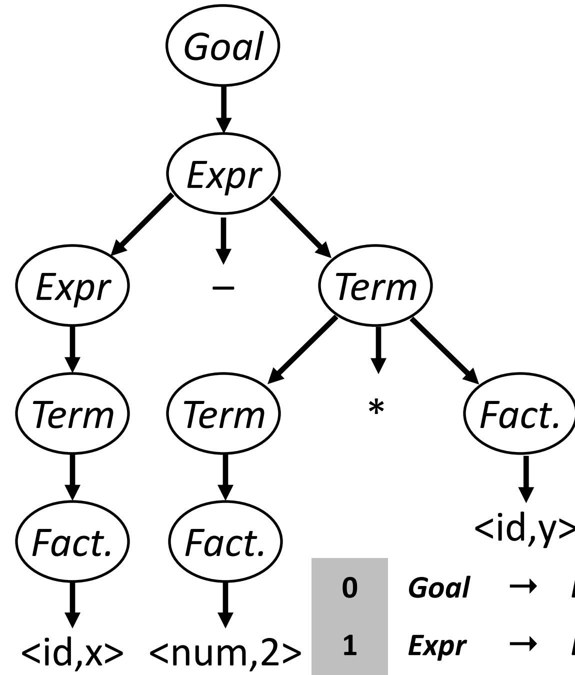
- “2” matches “2”
- We have more input, but no **NT**s left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

Trying again with "2" in x – 2 * y :

Rule	Sentential Form	Input
→	<id,x> - *Term*	x - ↑2 * y
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↑



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		|	identifier

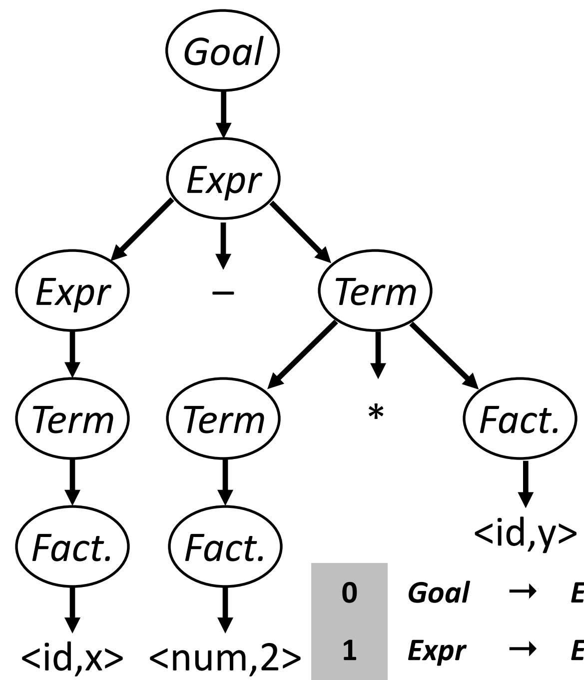
This time, we matched & consumed all the input

⇒Success!

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↑



0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* – *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
			actor
7	*Factor*	→	(*Expr*)
8		|	number
9		|	identifier

The Point:

For efficiency, the parser must make the correct choice when it expands an **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

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		|	identifier

The Classic 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

Classic Expression Grammar

The possibility of an infinite sequence of expansions in a parser is ~~bad~~. **disastrous**

- The problem arises from *left recursion* in the grammar and a *leftmost* derivation[1]
- **LHS** symbol cannot appear at start of the **RHS**
  – *Cannot derive from it in multiple steps, either*
- Top down parsers build leftmost derivations, so grammars with left recursion are not suitable for top-down parsing

[1] Similar problem arises with *right recursion* and a *rightmost* derivation.

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is **left recursive** if $\exists\ A \in NT$ such that a derivation $A \Rightarrow^+ A\alpha$ exists, for some string $\alpha \in (NT \cup T)^+$

Our classic expression grammar is left recursive

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

Non-termination is <u>always</u> a bad property in a compiler

Fortunately, we can eliminate left recursion in an algorithmic way.

Right recursion is defined in a symmetric way.

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

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

where neither α nor β start with *Fee*

Language is β followed by 0 or more α's

We can rewrite this fragment as

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

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

New Idea: the ε production

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	$\rightarrow$	*Expr + Term*	*Term*	$\rightarrow$	*Term * Factor*
	|	*Expr - Term*		|	*Term * Factor*
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	$\rightarrow$	*Term Expr'*	*Term*	$\rightarrow$	*Factor Term'*
Expr'	$\rightarrow$	*+ Term Expr'*	*Term'*	$\rightarrow$	** Factor Term'*
	|	*- Term Expr'*		|	*/ Factor Term'*
	|	ε		|	ε

These fragments use only right recursion

Eliminating Left Recursion

Substituting them back into the grammar yields

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	$\rightarrow$	<u>(</u> *Expr* <u>)</u>
10		|	<u>number</u>
11		|	<u>id</u>

Right-recursive expression grammar

- This grammar is correct, if somewhat counter-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- It is left associative, as was the original
  - ⇒ Why didn't we just rewrite it so *Expr* was at the right end of the **RHS**, rather than the left end?

Eliminating Left Recursion

Substituting them back into the grammar yields

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

Right-recursive expression grammar

- This grammar is correct, if

NOTE: This technique eliminates *direct left recursion* — when a production's RHS begins with its own LHS.

It does not eliminate *indirect left recursion*. We will get there, in the next lecture …

original

⇒ Why didn't we just rewrite it so *Expr* was at the right end of the **RHS**, rather than the left end?

Eliminating Left Recursion

Notice that we do not use the naïve right-recursive form

Expr	$\rightarrow$	*Term Expr'*	*Expr*	$\rightarrow$	*Term + Expr*
Expr'	$\rightarrow$	*+ Term Expr'*		|	*Term - Expr*
	|	*- Term Expr'*		|	*Term*
	|	ε			
		Transformed grammar fragment			**Naïve right-recursive form**

The naïve right-recursive form generates a different associativity (and parse tree) than did the original grammar.

The transformed grammar fragment generates the same parse tree as the original grammar did. (See § 3.5.3 in EaC2e.)

Eliminating Left Recursion

The naïve right-recursive form changes the associativity

$$\begin{array}{lcl} \mathit{Expr} & \rightarrow & \mathit{Term\ Expr'} \\ \mathit{Expr'} & \rightarrow & +\ \mathit{Term\ Expr'} \\ & | & -\ \mathit{Term\ Expr'} \\ & | & \varepsilon \end{array}$$

Transformed grammar fragment

$$\begin{array}{lcl} \mathit{Expr} & \rightarrow & \mathit{Term} + \mathit{Expr} \\ & | & \mathit{Term} - \mathit{Expr} \\ & | & \mathit{Term} \end{array}$$

Naïve right-recursive form

Abstracted Parse Trees for w + x + y + z

Parsing with the **RR CEG**

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

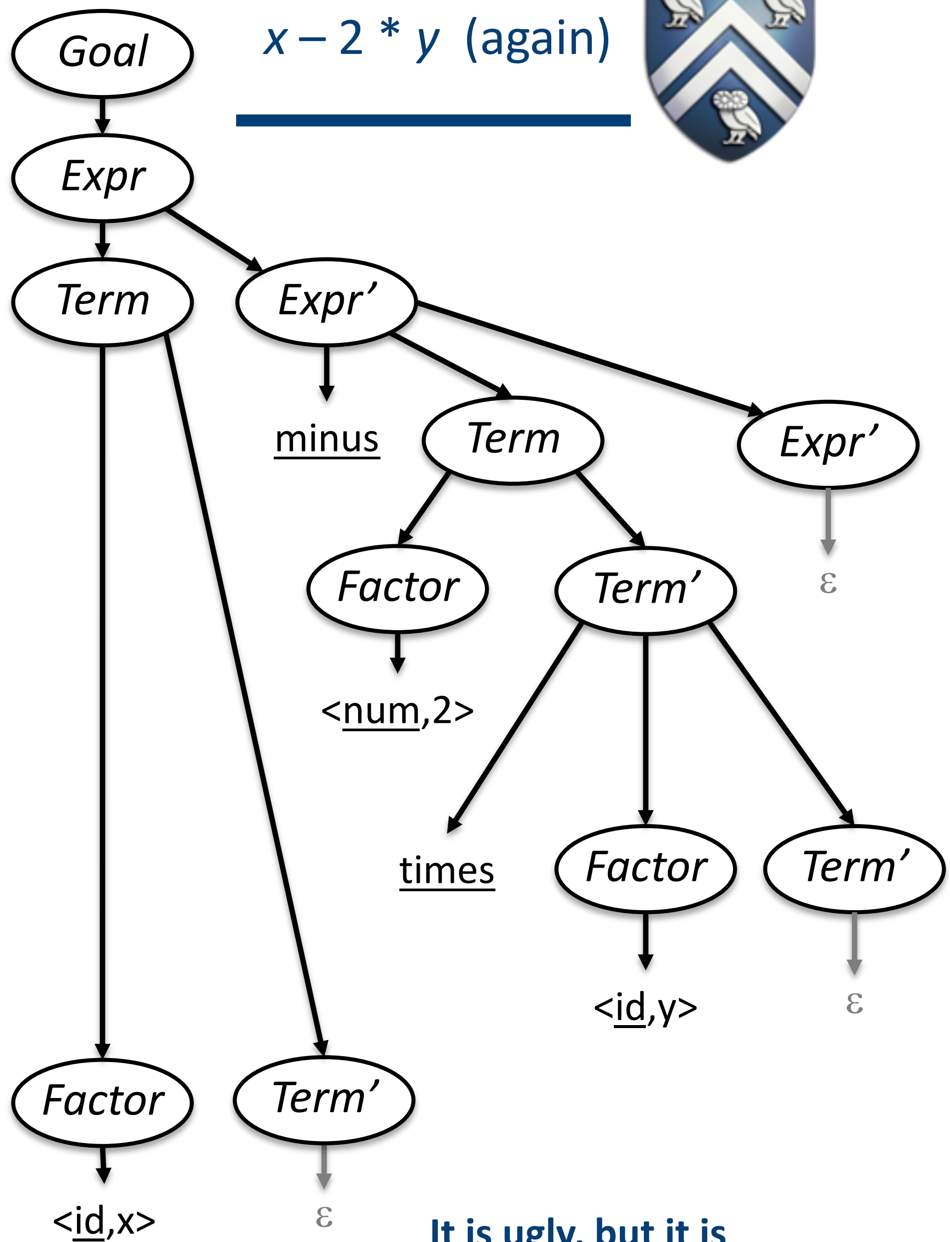
Right Recursive CEG

Rule	***Sentential Form*** **x – 2 * y, again**
—	*Goal*
0	*Expr*
1	*Term Expr'*
5	*Factor Term' Expr'*
11	<id,x> *Term' Expr'*
8	<id,x> *Expr'*
3	<id,x> - *Term Expr'*
5	<id,x> - *Factor Term' Expr'*
10	<id,x> - <num,2> *Term' Expr'*
6	<id,x> - <num,2> * *Factor Term' Expr'*
11	<id,x> - <num,2> * <id,y> *Term' Expr'*
8	<id,x> - <num,2> * <id,y> *Expr'*
4	<id,x> - <num,2> * <id,y>

Parsing with **RR CEG**

x – 2 * *y* (again)

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
1	*Term Expr'*
5	*Factor Term' Expr'*
11	<id,x> *Term' Expr'*
8	<id,x> *Expr'*
3	<id,x> - *Term Expr'*
5	<id,x> - *Factor Term' Expr'*
10	<id,x> - <num,2> *Term' Expr'*
6	<id,x> - <num,2> * *Factor Term' Expr'*
11	<id,x> - <num,2> * <id,y> *Term' Expr'*
8	<id,x> - <num,2> * <id,y> *Expr'*
4	<id,x> - <num,2> * <id,y>



It is ugly, but it is left associative.