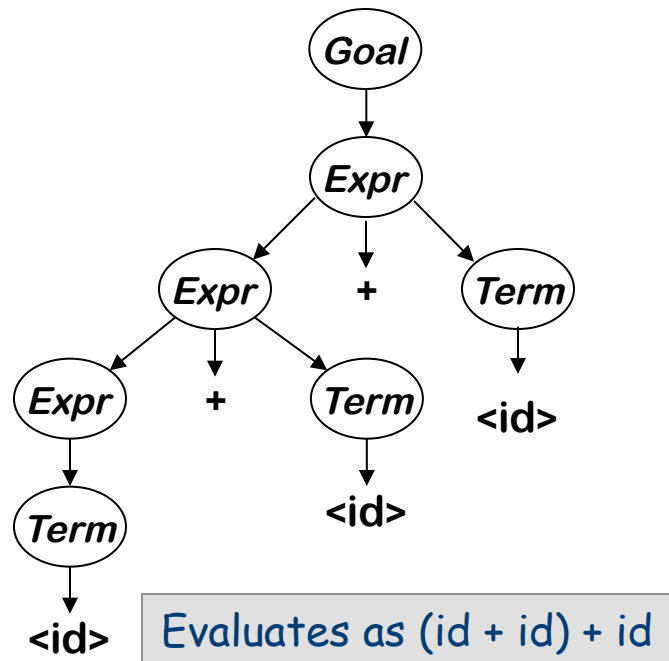




# Left Recursion Creates Left Associative Trees

A Trivial Expression Grammar

- 0 *Goal* → *Expr*
- 1 *Expr* → *Expr* + *Term*
- 2           | *Term*
- 3 *Term* → *id*



## Derivation of $id + id + id$

Rule	Sentential Form
—	<i>Goal</i>
0	<i>Expr</i>
1	<i>Expr</i> + <i>Term</i>
3	<i>Expr</i> + <u><i>id</i></u>
1	<i>Expr</i> + <i>Term</i> + <u><i>id</i></u>
3	<i>Expr</i> + <u><i>id</i></u> + <u><i>id</i></u>
2	<i>Term</i> + <u><i>id</i></u> + <u><i>id</i></u>
3	<u><i>id</i></u> + <u><i>id</i></u> + <u><i>id</i></u>



COMP 412  
FALL 2009

# *Top Down Parsing - Part I*

## *Comp 412*

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

# 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 lower 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?

---

We will call this version “the classic expression grammar”

— *from last lecture*

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

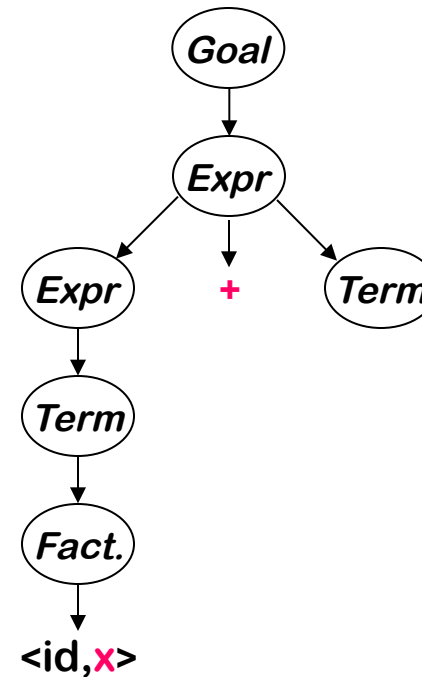
*And the input  $x - 2 * y$*



# Example

Let's try  $x - 2 * y$ :

Rule	Sentential Form	Input
—	Goal	$\uparrow x - 2 * y$
0	Expr	$\uparrow x - 2 * y$
1	Expr + Term	$\uparrow x - 2 * y$
3	Term + Term	$\uparrow x - 2 * y$
6	Factor + Term	$\uparrow x - 2 * y$
9	$\langle id, x \rangle + Term$	$\uparrow x - 2 * y$
→	$\langle id, x \rangle + Term$	$x \uparrow - 2 * y$



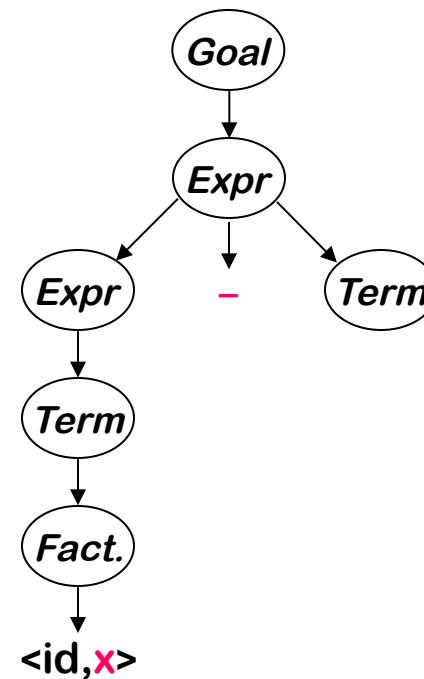
This worked well, except that "-" doesn't match "+"  
The parser must backtrack to here



# Example

Continuing with  $x - 2 * y$  :

Rule	Sentential Form	Input
—	<i>Goal</i>	$\uparrow x - 2 * y$
0	<i>Expr</i>	$\uparrow x - 2 * y$
<hr style="border-top: 1px dashed #ff00ff;"/>		
2	<i>Expr - Term</i>	$\uparrow x - 2 * y$
3	<i>Term - Term</i>	$\uparrow x - 2 * y$
6	<i>Factor - Term</i>	$\uparrow x - 2 * y$
9	$\langle id, x \rangle - Term$	$\uparrow x - 2 * y$
→	$\langle id, x \rangle \ominus Term$	$x \uparrow \ominus 2 * y$
→	$\langle id, x \rangle - Term$	$x - \uparrow 2 * y$



Now, "-" and "-" match

Now we can expand Term to match "2"

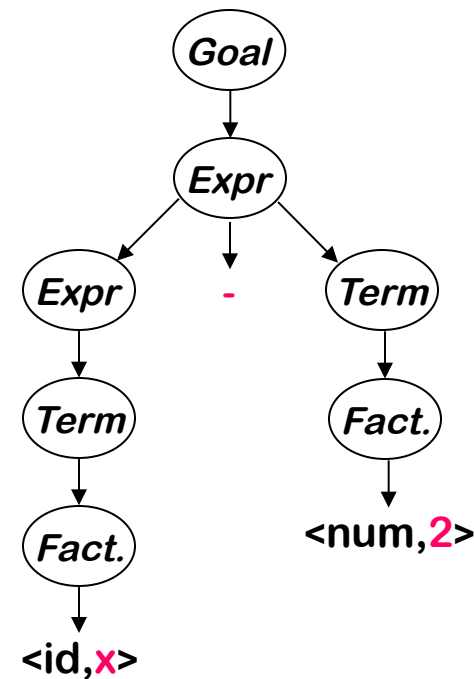
⇒ Now, we need to expand *Term* - the last *NT* on the fringe



# Example

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

Rule	Sentential Form	Input
→	$\langle id, x \rangle - Term$	$x - \uparrow 2 * y$
6	$\langle id, x \rangle - Factor$	$x - \uparrow 2 * y$
8	$\langle id, x \rangle - \langle num, 2 \rangle$	$x - \uparrow 2 * y$
→	$\langle id, x \rangle - \langle num, 2 \rangle$	$x - 2 \uparrow * y$



Where are we?

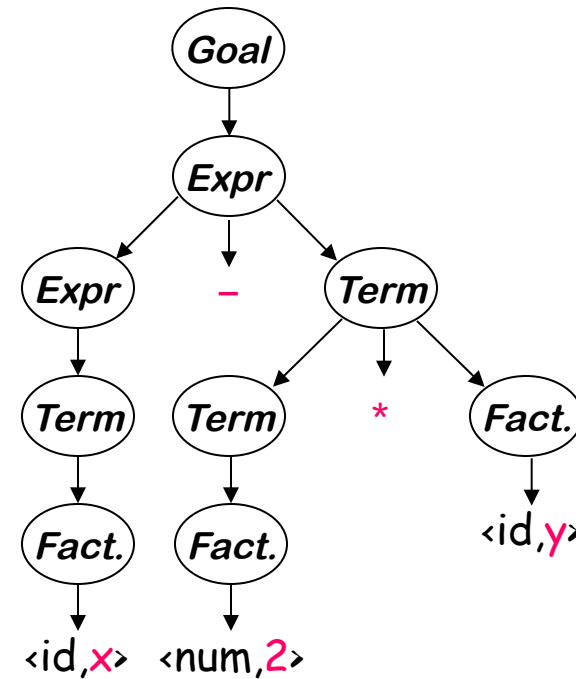
- "2" matches "2"
  - We have more input, but no *NTs* left to expand
  - The expansion terminated too soon
- ⇒ Need to backtrack



# Example

Trying again with "2" in  $x - 2 * y$  :

Rule	Sentential Form	Input
→	$\langle id, x \rangle - Term$	$x - \uparrow 2 * y$
4	$\langle id, x \rangle - Term * Factor$	$x - \uparrow 2 * y$
6	$\langle id, x \rangle - Factor * Factor$	$x - \uparrow 2 * y$
8	$\langle id, x \rangle - \langle num, 2 \rangle * Factor$	$x - \uparrow 2 * y$
→	$\langle id, x \rangle - \langle num, 2 \rangle * Factor$	$x - 2 \uparrow * y$
→	$\langle id, x \rangle - \langle num, 2 \rangle * Factor$	$x - 2 * \uparrow y$
9	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$	$x - 2 * \uparrow y$
→	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$	$x - 2 * y \uparrow$



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	$\uparrow x - 2 * y$
0	Expr	$\uparrow x - 2 * y$
1	Expr + Term	$\uparrow x - 2 * y$
1	Expr + Term + Term	$\uparrow x - 2 * y$
1	Expr + Term + Term + Term	$\uparrow x - 2 * y$
1	And so on ....	$\uparrow x - 2 * y$

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

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 always a bad property in a compiler*



# Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

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

where neither  $\alpha$  nor  $\beta$  start with  $Fee$

We can rewrite this fragment as

$$\begin{aligned} Fee &\rightarrow \beta Fie \\ Fie &\rightarrow \alpha Fie \\ &| \epsilon \end{aligned}$$

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

$$\begin{array}{l} \text{Expr} \rightarrow \text{Expr} + \text{Term} \\ \quad | \text{Expr} - \text{Term} \\ \quad | \text{Term} \end{array} \qquad \begin{array}{l} \text{Term} \rightarrow \text{Term} * \text{Factor} \\ \quad | \text{Term} / \text{Factor} \\ \quad | \text{Factor} \end{array}$$

Applying the transformation yields

$$\begin{array}{l} \text{Expr} \rightarrow \text{Term Expr}' \\ \text{Expr}' \rightarrow + \text{Term Expr}' \\ \quad | - \text{Term Expr}' \\ \quad | \varepsilon \end{array} \qquad \begin{array}{l} \text{Term} \rightarrow \text{Factor Term}' \\ \text{Term}' \rightarrow * \text{Factor Term}' \\ \quad | / \text{Factor Term}' \\ \quad | \varepsilon \end{array}$$

These fragments use only right recursion

Right recursion often means right associativity. In this case, the grammar does not display any particular associative bias.<sub>12</sub>



# Eliminating Left Recursion

Substituting them back into the grammar yields

0	<i>Goal</i>	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Term Expr'</i>
2	<i>Expr'</i>	→	+ <i>Term Expr'</i>
3			- <i>Term Expr'</i>
4			$\epsilon$
5	<i>Term</i>	→	<i>Factor Term'</i>
6	<i>Term'</i>	→	* <i>Factor Term'</i>
7			/ <i>Factor Term'</i>
8			$\epsilon$
9	<i>Factor</i>	→	( <i>Expr</i> )
10			<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  $A_1, A_2, \dots, A_n$*

*for  $i \leftarrow 1$  to  $n$*

*for  $s \leftarrow 1$  to  $i - 1$*

*replace each production  $A_i \rightarrow A_s \gamma$  with  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ ,*

*where  $A_s \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  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



## 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.  $A_i = G$

$G \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow E \sim T$

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

2.  $A_i = E$

$G \rightarrow E$

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$

$T \rightarrow E \sim T$

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

3.  $A_i = T, A_s = E$

$G \rightarrow E$

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$

$T \rightarrow TE' \sim T$

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

4.  $A_i = T$

$G \rightarrow E$

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$

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

$T' \rightarrow E \sim TT'$

$T' \rightarrow \varepsilon$

# 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 lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

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

# Predictive Parsing

---



## Basic idea

*Given  $A \rightarrow \alpha \mid \beta$ , the parser should be able to choose between  $\alpha$  &  $\beta$*

## FIRST sets

For some *rhs*  $\alpha \in G$ , define  $\text{FIRST}(\alpha)$  as the set of tokens that appear as the first symbol in some string that derives from  $\alpha$

That is,  $\underline{x} \in \text{FIRST}(\alpha)$  iff  $\alpha \Rightarrow^* \underline{x}\gamma$ , for some  $\gamma$

We will defer the problem of how to compute FIRST sets for the moment.

# Predictive Parsing



## Basic idea

Given  $A \rightarrow \alpha \mid \beta$ , the parser should be able to choose between  $\alpha$  &  $\beta$

## FIRST sets

For some rhs  $\alpha \in G$ , define  $\text{FIRST}(\alpha)$  as the set of tokens that appear as the first symbol in some string that derives from  $\alpha$

That is,  $\underline{x} \in \text{FIRST}(\alpha)$  iff  $\alpha \Rightarrow^* \underline{x}\gamma$ , for some  $\gamma$

## The LL(1) Property

If  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct  
See the next slide



# Predictive Parsing

---

What about  $\epsilon$ -productions?

$\Rightarrow$  They complicate the definition of  $LL(1)$

If  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  and  $\epsilon \in \text{FIRST}(\alpha)$ , then we need to ensure that  $\text{FIRST}(\beta)$  is disjoint from  $\text{FOLLOW}(A)$ , too, where

$\text{FOLLOW}(A)$  = the set of terminal symbols that can immediately follow  $A$  in a sentential form

Define  $\text{FIRST}^+(A \rightarrow \alpha)$  as

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$ , if  $\epsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$ , otherwise

Then, a grammar is  $LL(1)$  iff  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  implies

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$$





# Predictive Parsing

Given a grammar that has the  $LL(1)$  property

- Can write a simple routine to recognize each *lhs*
- Code is both simple & fast

Consider  $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$ , with

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset \text{ if } i \neq j$$

```
/* find an A */  
if (current_word ∈ FIRST(A → β1))  
    find a β1 and return true  
else if (current_word ∈ FIRST(A → β2))  
    find a β2 and return true  
else if (current_word ∈ FIRST(A → β3))  
    find a β3 and return true  
else  
    report an error and return false
```

Grammars with the  $LL(1)$  property are called predictive grammars because the parser can “predict” the correct expansion at each point in the parse.

Parsers that capitalize on the  $LL(1)$  property are called predictive parsers.

One kind of predictive parser is the recursive descent parser.

Of course, there is more detail to “find a  $\beta_i$ ” (p. 103 in EAC, 1<sup>st</sup> Ed.)

# Recursive Descent Parsing



Recall the expression grammar, after transformation

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

This produces a parser with six mutually recursive routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term descent refers to the direction in which the parse tree is built.

# Recursive Descent Parsing

# (Procedural)



A couple of routines from the expression parser

## Goal()

```
token ← next_token();  
if (Expr() = true & token = EOF)  
    then next compilation step;  
else  
    report syntax error;  
    return false;
```

## Expr()

```
if (Term() = false)  
    then return false;  
else return Eprime();
```

looking for Number, Identifier, or  
"(", found token instead, or failed  
to find Expr or ")" after "("

## Factor()

```
if (token = Number) then  
    token ← next_token();  
    return true;  
else if (token = Identifier) then  
    token ← next_token();  
    return true;  
else if (token = Lparen)  
    token ← next_token();  
    if (Expr() = true & token = Rparen) then  
        token ← next_token();  
        return true;  
    // fall out of if statement  
    report syntax error;  
    return false;
```

*EPrime, Term, & TPrime follow the same  
basic lines (Figure 3.7, EAC)*