COMP 412
FALL 2010

Top Down Parsing - Part I

Comp 412

Copyright 2010, 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	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	$\rightarrow$	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	$\rightarrow$	(*Expr*)
8		|	number
9		|	id

And the input x – 2 * y

Example

Let's try x – 2 * y :

Goal

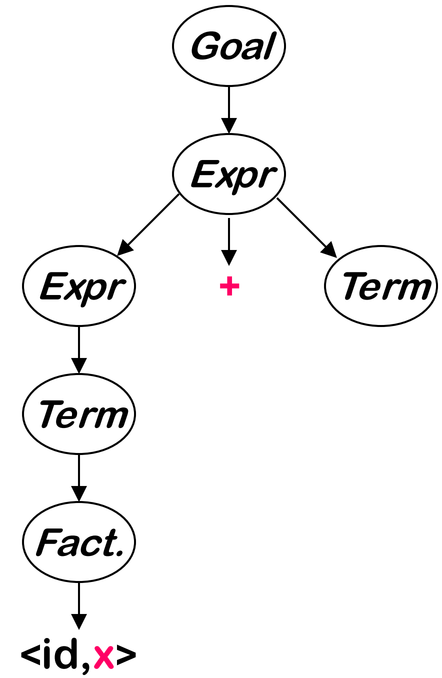
Rule	Sentential Form	Input
—	Goal	↑x - 2 * y

↑ 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 "+"

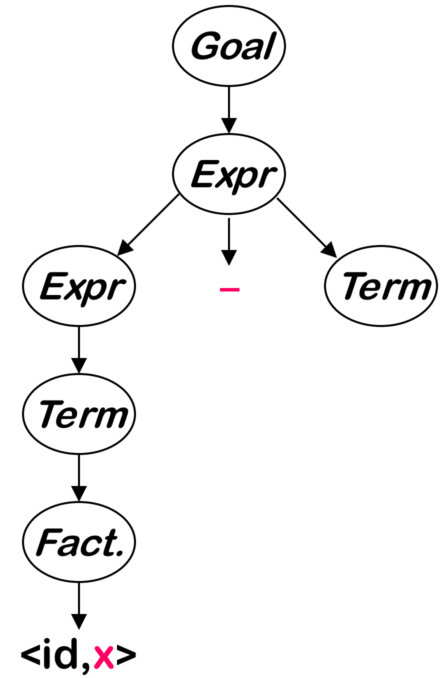
The parser must backtrack to here

↑ *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

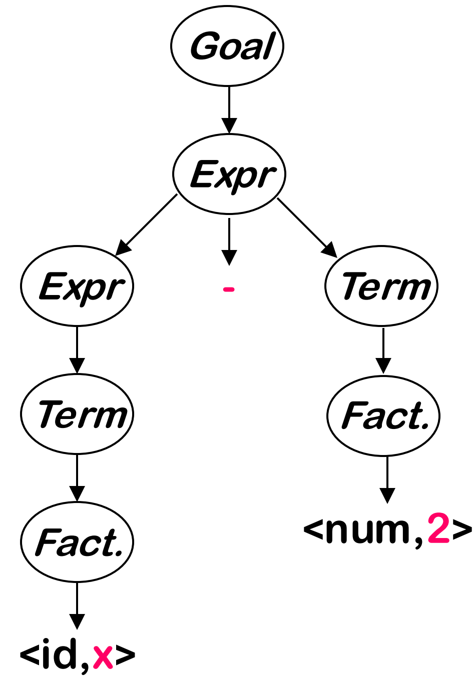
Now we can expand Term to match "2"

⇒ 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?

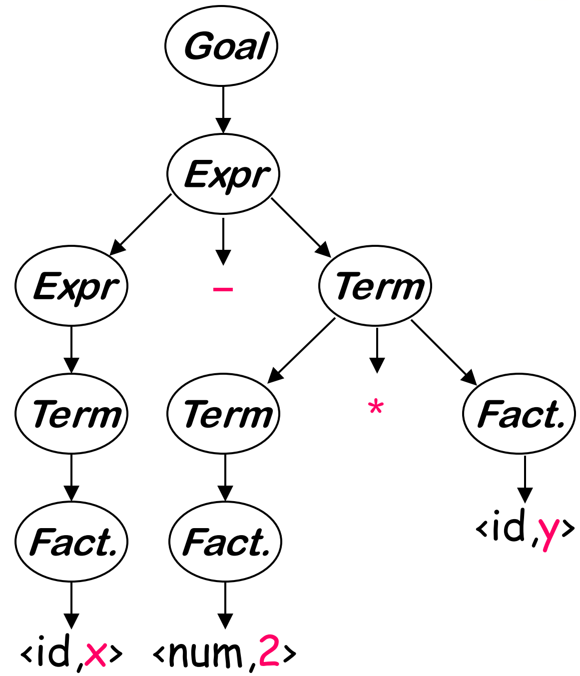
- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ 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 $\alpha \in (NT \cup T)^{+}$

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

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$Fee \rightarrow Fee\ \alpha$$
$$\quad\quad\mid\ \beta$$

where neither α nor β start with *Fee*

We can rewrite this fragment as

$$Fee \rightarrow \beta\ Fie$$
$$Fie \rightarrow \alpha\ Fie$$
$$\quad\quad\mid\ \varepsilon$$

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

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

Applying the transformation yields

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

These fragments use only right recursion

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

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

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
  - ⇒ The naïve transformation yields a right recursive grammar, which changes the implicit associativity
- 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

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

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. $A_i = G$	2. $A_i = E$	3. $A_i = T$, $A_s = E$	4. $A_i = T$
$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$
$E \rightarrow E + T$	$E \rightarrow T\, E'$	$E \rightarrow T\, E'$	$E \rightarrow T\, E'$
$E \rightarrow T$	$E' \rightarrow +\, T\, E'$	$E' \rightarrow +\, T\, E'$	$E' \rightarrow +\, T\, E'$
$T \rightarrow E * T$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
$T \rightarrow \underline{\text{id}}$	$T \rightarrow E * T$	$T \rightarrow T\, E' * T$	$T \rightarrow \underline{\text{id}}\; T'$
	$T \rightarrow \underline{\text{id}}$	$T \rightarrow \underline{\text{id}}$	$T' \rightarrow E' * T\, T'$
			$T' \rightarrow \varepsilon$

Go to Algorithm

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

We will focus, for now, on LL(1) grammars & predictive parsing

Predictive Parsing

Basic idea

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

FIRST sets

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

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

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 α & β

FIRST sets

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

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

The LL(1) Property

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

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

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 ε-productions?

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

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), too, where

FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form

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

- FIRST(α) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies FIRST^{+}($A \rightarrow \alpha$) $\cap$ 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 β_i" (p. 103 in EAC, 1st Ed.)

Recursive Descent Parsing

Recall the expression grammar, after transformation

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

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)

Extra Slides Start Here

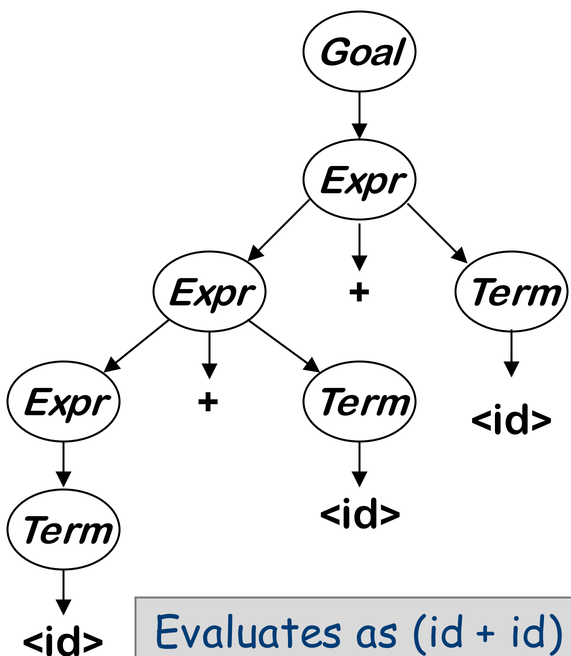
Left Recursion Creates Left Associative Trees

A Trivial Expression Grammar

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

Derivation of id + id +id

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
1	*Expr* + *Term*
3	*Expr* + <*id*>
1	*Expr* + *Term* + <id>
3	*Expr* + <id> + <id>
2	*Term* + <id> + <id>
3	<id> + <id> + <id>



Evaluates as (id + id) + id