Lecture 14: LL Parsers

Gul Agha

(primarily based on slides by Mattox Beckman
as modified by Vikram Adve)

agha@cs.uiuc.edu

University of Illinois
Objectives

This lecture covers a class of top-down parsing algorithms, LL(k) parsing, and a specific example of this class, *recursive-descent parsing*. An R-D parser may be the easiest to write for a simple grammar, and R-D has been used even for complex languages like C++.

Your goals for this lecture:

- Know how to tell if a grammar is LL.
- Know how to write a recursive descent LL(1) parser.
- Know how to detect and eliminate left recursion.
- Know how to detect and eliminate common prefixes.

Further reading: See Dragon Book §4.x
What is LL(n) Parsing?

- An LL parse uses a **Left-to-right scan** and produces a **Leftmost derivation**, using **n** tokens of lookahead.
- **A.K.A. Top-Down Parsing**

**Example Grammar:**

\[
S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow \ast \ E \ E
\]

**Example Input:** \(+\ 2 \ \ast \ 3 \ 4\)

**Syntax Tree:**

```
  S
 / \  \\
/   \\
S   \\
```

**Derivations:**

```
S
 S
```

---

University of Illinois
What is LL(n) Parsing?

- An LL parse uses a **Left-to-right** scan and produces a **Leftmost** derivation, using **n** tokens of lookahead.

- A.K.A. Top-Down Parsing

**Example Grammar:**

\[
S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow * \ E \ E
\]

**Example Input:** + 2 * 3 4

**Syntax Tree:**

```
     S
     /|
    + E
    /|
   E E
```

**Derivations:**

```
S
+ E E
```
What is LL(n) Parsing?

- An LL parse uses a Left-to-right scan and produces a Leftmost derivation, using n tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

\[ S \rightarrow + \ E \ E \]
\[ E \rightarrow \text{int} \]
\[ E \rightarrow * \ E \ E \]

Example Input: + 2 * 3 4

Syntax Tree:

```
       S
      /|
     + E
    / |
   + 2
```

Derivations:

- \[ S \]
- \[ + E E \]
- \[ + 2 E \]
What is LL(n) Parsing?

- An LL parse uses a Left-to-right scan and produces a Leftmost derivation, using n tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

\[ S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow * \ E \ E \]

Example Input: \(+ 2 \times 3 4\)

Syntax Tree:

```
   S
  /|
 / |
/  |
E + E
  /  |
 /   |
2 * E
```

Derivations:

- \( S \)
- \( + \ E \ E \)
- \( + 2 \ E \)
- \( + 2 \times E \ E \)
What is LL(n) Parsing?

- An LL parse uses a **Left-to-right scan** and produces a **Leftmost derivation**, using **n** tokens of lookahead.

- A.K.A. Top-Down Parsing

**Example Grammar:**

\[
S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow * \ E \ E
\]

**Example Input:**  
\[+ \ 2 \ * \ 3 \ 4\]

**Syntax Tree:**

\[
\text{S} \quad + \quad \text{E} \quad \text{E} \\
\quad + \quad \quad \text{E} \\
\quad \quad \text{2} \\
\quad \quad * \\
\quad \quad \text{E} \\
\quad \quad \quad \text{3} \\
\quad \quad \quad \text{E} \\
\quad \quad \quad \text{E}
\]

**Derivations:**

\[
S \\
+ \ E \ E \\
+ \ 2 \ E \\
+ \ 2 \ * \ E \ E \\
+ \ 2 \ * \ 3 \ E
\]
What is LL(n) Parsing?

- An LL parse uses a Left-to-right scan and produces a Leftmost derivation, using n tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

\[
S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow * \ E \ E
\]

Example Input: \(+\ 2 \ * \ 3 \ 4\)

Syntax Tree:

\[
\begin{array}{c}
S \\
+ \\
2 \\
* \\
E \\
3 \\
E \\
4 \\
\end{array}
\]

Derivations:

\[
S \\
+ \ E \ E \\
+ \ 2 \ E \\
+ \ 2 \ * \ E \ E \\
+ \ 2 \ * \ 3 \ E \\
+ \ 2 \ * \ 3 \ 4
\]
How to Write a Recursive Descent Parser

Recursive Descent Parsers: One kind of LL parser

- Recursively construct the parse tree top-down: roughly one function call per internal node

- Are very easy to implement: no tables here!
  Directly translate grammar into recursive code

First, create data types to represent your syntax tree. See example for lambda expressions in text. (covered in an earlier lecture).
How to Write a Recursive Descent Parser

- Compute FIRST sets (explanation later)
- Write one function for each non-terminal:
  - Implements all rules for the non-terminal
  - Uses lookahead token(s), if any, to choose the rule
- Each function has type
  ```
  string list -> tree * string list
  ```
  - input: list of tokens
  - output: sub-tree of parse tree, plus remaining tokens.
- Each terminal matched in a rule consumes an input (token)
Things to Notice

- Each function immediately checks the first token of the input string. (LL(0) parsers will consume the token immediately.)
- This token is used to decide what to do next.
- What kinds of things could go wrong?
A rule like $E \rightarrow E + E$ would cause an infinite loop.
Rules with Common Prefixes

A rule like $E \rightarrow -E \mid -EE$ would confuse the function. Which version of the rule should be used?
The Idea

Consider deriving $i++++$ from the following grammar:

\[
E 
\]

Derivation:

\[
E \rightarrow E + E \\
E \rightarrow i
\]
The Idea

Consider deriving $E \rightarrow i$ from the following grammar:

**Derivation:**

$$E \rightarrow E +$$
$$E \rightarrow i$$

**Diagram:**

```
  E
 /\  \
/   \
E +  E
```
The Idea

Consider deriving $i++++$ from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$

Derivation:

E
E+
E++
The Idea

Consider deriving $i++++$ from the following grammar:

$$E \rightarrow E +$$
$$E \rightarrow i$$

Derivation:

1. $E$
2. $E +$
3. $E++$
4. $E++++$
The Idea

Consider deriving $i++++$ from the following grammar:

$E \rightarrow E +$

$E \rightarrow i$

Derivation:

$E$
$E+$
$E++$
$E++++$
$E+++++$
The Idea

Consider deriving $i++++$ from the following grammar:

$$E \rightarrow E +$$
$$E \rightarrow i$$

Derivation:

$$E$$
$$E+$$
$$E++$$
$$E++++$$
$$E+++++$$
$$i++++$$
The Idea

Consider deriving $i++++$ from the following grammar:

$E \rightarrow E +$
$E \rightarrow i$

Derivation:

```
E
E+
E++
E+++ 
E++++ 
i++++
```

- The rule $E \rightarrow E+$ says that we can have as many $+$s as we want at the end of the sentence.
- The rule $E \rightarrow i$ says—in effect—that the first word can be a $i$.
- Question: isn’t there another way to write this?
More complicated example

Consider the following grammar. What does it mean?

\[ B \rightarrow Bxy \mid Bz \mid q \mid r \]

- At the end can come any combination of \( x \), \( y \) or \( z \)
- At the beginning can come \( q \) or \( r \)
Eliminating the Left Recursion

We can rewrite these grammars

\[ E \rightarrow E + | i \]
\[ B \rightarrow Bxy | Bz | q | r \]

using the following transformation:

- Productions of the form \( S \rightarrow \beta \) become \( S \rightarrow \beta S' \).
- Productions of the form \( S \rightarrow S\alpha \) become \( S' \rightarrow \alpha S' \).
- Add \( S' \rightarrow \epsilon \).

\[ E \rightarrow iE' \]
\[ E' \rightarrow +E' | \epsilon \]
\[ B \rightarrow qB' | rB' \]
\[ B' \rightarrow xyB' | zB' | \epsilon \]

Result:
Mutual Recursions!

Things are slightly more complicated if we have mutual recursions.

\[ A \rightarrow Aa \mid Bb \mid Cc \mid q \]
\[ B \rightarrow Ax \mid By \mid Cz \mid rA \]
\[ C \rightarrow Ai \mid Bj \mid Ck \mid sB \]

How to do it:

- Take the first symbol (A) and eliminate immediate left recursion.
- Take the second symbol (B), and substitute left recursions to A. Then eliminate immediate left recursion in B.
- Take the third symbol (C) and substitute left recursions to A and B. Then eliminate immediate left recursion in C.
Left Recursion Example

Here is a more complex left recursion.

\[
\begin{align*}
A & \rightarrow Aa \mid Bb \mid Cc \mid q \\
B & \rightarrow Ax \mid By \mid Cz \mid rA \\
C & \rightarrow Ai \mid Bj \mid Ck \mid sB
\end{align*}
\]

First we eliminate the left recursion from \( A \).

\[
A \rightarrow Aa \mid Bb \mid Cc \mid q
\]

becomes

\[
A \rightarrow BbA' \mid CcA' \mid qA'
A' \rightarrow aA' \mid \epsilon
\]
Substituting in the new definition of $A$, and now we will work on the $B$ productions.

\[
A \to BbA' \mid CcA' \mid qA' \\
A' \to aA' \mid \epsilon \\
\boxed{B \to Ax \mid By \mid Cz \mid rA} \\
C \to Ai \mid Bj \mid Ck \mid sB
\]

First, we eliminate the “backward” recursion from $B$ to $A$.

\[
B \to Ax \text{ becomes} \\
B \to BbA'x \mid CcA'x \mid qA'x
\]
Left Recursion Example, 3

\[ A \rightarrow BbA' \mid CcA' \mid qA' \]
\[ A' \rightarrow aA' \mid \epsilon \]
\[ B \rightarrow BbA'x \mid CcA'x \mid qA'x \mid By \mid Cz \mid rA \]
\[ C \rightarrow Ai \mid Bj \mid Ck \mid sB \]

Now we can eliminate the simple left recursion in \( B \), to get
\[ B \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \]
\[ B' \rightarrow bA'xB' \mid yB' \mid \epsilon \]
Left Recursion Example, 4

\[
A \rightarrow BbA' \mid CcA' \mid qA' \\
A' \rightarrow aA' \mid \epsilon \\
B \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \\
B' \rightarrow bA'xB' \mid yB' \mid \epsilon \\
C \rightarrow Ai \mid Bj \mid Ck \mid sB
\]

Now production \( C \): first, replace left recursive calls to \( A \)...

\[
C \rightarrow \boxed{B}bA'i \mid CcA'i \mid qA'i \mid \boxed{B}j \mid Ck \mid sB
\]

Next, replace left recursive calls to \( B \) (this gets messy)...

\[
C \rightarrow \boxed{CcA'xB'}bA'i \mid \boxed{qA'xB'}bA'i \mid \boxed{CzB'}bA'i \mid \boxed{rAB'}bA'i \\
\boxed{CcA'xB'}j \mid \boxed{qA'xB'}j \mid \boxed{CzB'}j \mid \boxed{rAB'}j \\
CcA'i \mid qA'i \mid Ck \mid sB
\]
Left Recursion Example, 5

Reorganizing $C$, we have

\[ C \rightarrow qA'xB'bA'i | rAB'bA'i | qA'xB'j | rAB'j | qA'i | sB \\
| CaA'xB'bA'i | CaA'xB'j | CzB'j | CcA'i | Ck \]

Eliminating left recursion gives us

\[ C \rightarrow qA'xB'bA'iC' | rAB'bA'iC' | qA'xB'jC' | rAB'jC' | qA'iC' | sB \\
C' \rightarrow cA'xB'bA'iC' | zB'bA'iC' | cA'xB'jC' | zB'jC' | cA'iC' | kC' | \epsilon \]
The result...

Our final grammar is now

\[
\begin{align*}
A & \rightarrow BbA' \mid CcA' \mid qA' \\
A' & \rightarrow aA' \mid \epsilon \\
B & \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \\
B' & \rightarrow bA'xB' \mid yB' \mid \epsilon \\
C & \rightarrow qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rAB'jC' \mid qA'iC' \mid sBC \\
C' & \rightarrow cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon
\end{align*}
\]

Beautiful, isn’t it? I wonder why we don’t do this more often?

 Disclaimer: if there is a cycle \((A \rightarrow^+ A)\) or an epsilon production \((A \rightarrow \epsilon)\) then this technique is not guaranteed to work.
Common Prefix

This grammar has common prefixes.

\[ A \rightarrow xyB \mid CyC \mid q \]
\[ B \rightarrow zC \mid zx \mid w \]
\[ C \rightarrow y \mid x \]

To check for common prefixes, take a non-terminal and compare the First sets of each production.

<table>
<thead>
<tr>
<th>Production</th>
<th>FirstSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ A \rightarrow xyB ]</td>
<td>{x}</td>
</tr>
<tr>
<td>[ A \rightarrow CyC ]</td>
<td>{x, y}</td>
</tr>
<tr>
<td>[ A \rightarrow q ]</td>
<td>{q}</td>
</tr>
</tbody>
</table>

If we are viewing an \( A \), we will want to look at the next token to see which \( A \) production to use. If that token is \( x \), then which production do we use?
Left Factoring

If \( A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \gamma \) we can rewrite it as

\[
A \rightarrow \alpha A' \mid \gamma \\
A' \rightarrow \beta_1 \mid \beta_2
\]

So, in our example:

\[
A \rightarrow xyB \mid CyC \mid q \quad \text{becomes} \quad A \rightarrow xA' \mid q \mid yyC' \\
B \rightarrow zC \mid zx \mid w \quad \qquad \quad A' \rightarrow yB \mid yC \\
C' \rightarrow y \mid x \quad \qquad \quad B \rightarrow zB' \mid w \\
\quad \qquad \quad B' \rightarrow C \mid x \\
\quad \qquad \quad C \rightarrow y \mid x
\]

Sometimes you’ll need to do this more than once. Note that this process can destroy the meaning of the nonterminals.
Activity

One of these is LL, the other two are not. Fix the ones that are not.

<table>
<thead>
<tr>
<th>Grammar 1</th>
<th>Grammar 2</th>
<th>Grammar 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \to E \times y )</td>
<td>( S \to A \times )</td>
<td>( S \to A \times )</td>
</tr>
<tr>
<td>( E \to E \times B )</td>
<td>( S \to B \times )</td>
<td>( S \to B \times )</td>
</tr>
<tr>
<td>( E \to q )</td>
<td>( A \to z B )</td>
<td>( A \to z B )</td>
</tr>
<tr>
<td>( B \to E \ z )</td>
<td>( B \to w A )</td>
<td>( B \to z )</td>
</tr>
</tbody>
</table>
Grammar 1

Grammar 1 starts as:

\[ E \rightarrow E \ x \ y \]
\[ E \rightarrow E \ x \ B \]
\[ E \rightarrow q \]
\[ B \rightarrow E \ z \]

Eliminate the left-recursion to get:

\[ E \rightarrow qE' \]
\[ E' \rightarrow x \ y \ E' \]
\[ E' \rightarrow x \ B \ E' \]
\[ E' \rightarrow \epsilon \]
\[ B \rightarrow E \ z \]

Fix the common prefixes to get:

\[ E \rightarrow qE' \]
\[ E' \rightarrow x \ E'' \]
\[ E' \rightarrow \epsilon \]
\[ E'' \rightarrow y \ E' \ | \ B \ E' \]
\[ B \rightarrow E \ z \]
Grammar 2

\[
\begin{align*}
S &\rightarrow A \ x \\
S &\rightarrow B \ y \\
A &\rightarrow z \ B \\
B &\rightarrow w \ A
\end{align*}
\]

It doesn't terminate. But it's not left recursive, and it has no productions with common prefixes, so it's still LL.
Grammar 3

\[ S \rightarrow A \ x \]
\[ S \rightarrow B \ y \]
\[ A \rightarrow z \ B \]
\[ B \rightarrow z \]

Production \( S \) has common prefixes. One way to fix this grammar is to eliminate the distinctions between \( A \) and \( B \) — this may not be what you want, though.

\[ S \rightarrow z \ S' \]
\[ S' \rightarrow z \ x \]
\[ S' \rightarrow y \]
Lambda-Calculus Concrete Syntax

1. `<expression>  -->
2.    <identifier>
3.  | (lambda (<identifier>)<expression>)
4.  | (<expression> <expression>)
Lambda-Calculus Expressions Data Type

```
(define-datatype expression expression
  (var-exp
    (id symbol?))
  (lambda-exp
    (id symbol?)
    (body expression?))
  (app-exp
    (rator expression?)
    (rand expression?))
```

University of Illinois
Parsing Lambda-Calculus Expressions

(define parse-expression
  (lambda (datum)
    (cond
     ((symbol? datum) (var-exp datum))
     ((pair? datum)
      (if (eqv? (car datum) 'lambda)
       (lambda-exp (caadr datum)
         (parse-expression (caddr datum)))
       (app-exp
         (parse-expression (car datum))
         (parse-expression (cadr datum)))))
     (else (eopl: error 'parse-expression
                     "Invalid concrete syntax ~s" datum)))))
Unparsing – Why?

```
(define unparse-expression
 (lambda (exp)
   (cases expression exp
     (var-exp (id) id)
     (lambda-exp (id body)
       (list 'lambda (list id
         (unparse-expression body)))
     (app-exp (rator rand)
       (list (unparse-expression rator)
         (unparse-expression rand))))))
```