LL Grammars

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

The topic for this lecture is a kind of grammar that works well with recursive-descent parsing.

- Know how to tell if a grammar is LL.
- Know what parsing technique will work with an LL grammar.
- Know how to detect and eliminate left recursion.
- Know how to detect and eliminate common prefixes.

What is LL(n) Parsing?

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

Example Grammar:

$$S \longrightarrow + \; E \; E$$
$$E \longrightarrow \texttt{int}$$
$$E \longrightarrow * \; E \; E$$

Syntax Tree:

S

Example Input:

`+ 2 * 3 4`

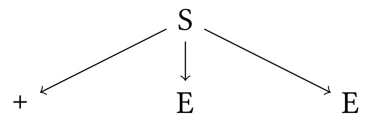
What is LL(n) Parsing?

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

Example Grammar:

$$S \longrightarrow + \; E \; E$$
$$E \longrightarrow \texttt{int}$$
$$E \longrightarrow * \; E \; E$$

Syntax Tree:



Example Input:

`+ 2 * 3 4`

What is LL(n) Parsing?

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

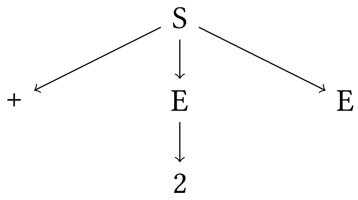
Example Grammar:

$S \rightarrow + \; E \; E$
$E \rightarrow \texttt{int}$
$E \rightarrow * \; E \; E$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



What is LL(n) Parsing?

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

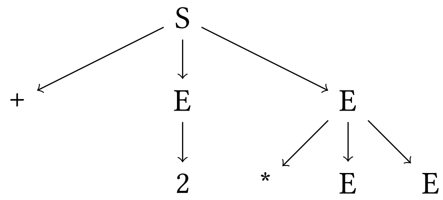
Example Grammar:

$S \rightarrow + \; E \; E$
$E \rightarrow \texttt{int}$
$E \rightarrow * \; E \; E$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



What is LL(n) Parsing?

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

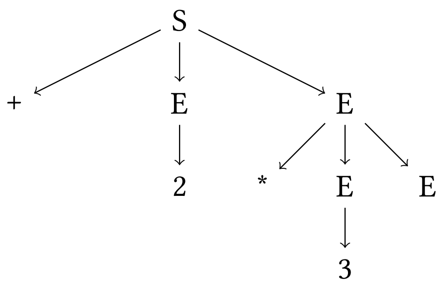
Example Grammar:

$S \rightarrow + \; E \; E$
$E \rightarrow \texttt{int}$
$E \rightarrow * \; E \; E$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



What is LL(n) Parsing?

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

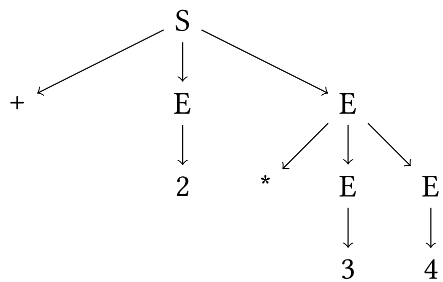
Example Grammar:

$S \rightarrow + \; E \; E$
$E \rightarrow \texttt{int}$
$E \rightarrow * \; E \; E$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



How to Implement It

Interpreting a Production

- Think of a production as a function definition.
- The LHS is the function being defined.
- Terminals on RHS are commands to consume input.
- Non-terminals on RHS are subroutine calls.

- For each production, make a function of type `[String] -> (Tree,[String])`
  - input is a list of tokens
  - output is a syntax tree and remaining tokens.
- Of course, you need to create a type to represent your tree.

Things to Notice

Key Point — Prediction

- Each function immediately checks the first token of the input string to see what to do next.

```
getE [] = undefined
getE ('*':xs) =
  let e1,r1 = getE xs
      e2,r2 = getE r1
   in (ETimes e1 e2, r2)
getE .... -- other code follows
```


Left Recursion

Left Recursion is Bad

- A rule like $E \rightarrow E + E$ would cause an infinite loop.

```
getE xx =
  let e1,r1 = getE xx
      ('+':r2) = r1
      e2,r3 = getE r2
   in (EPlus e1 e2, r3)
```


Rules with Common Prefixes

Common Prefixes are Bad

- A pair of rules rule like $\begin{array}{rl} E \rightarrow & - E \\ | & - E\ E \end{array}$ would confuse the function. Which version of the rule should be used?

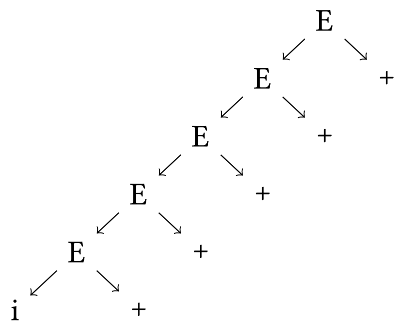
```
getE ('-':xs) = ... -- unary rule
getE ('-':xs) = ... -- binary rule
```

- NB: Common prefixes must be for the *same* non-terminal. E.g., $E \rightarrow x\ A$ and $S \rightarrow x\ B$ do not count as common prefixes.

The Idea

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

$E \to E+$	"We can have as many +s as we want *at the end* of the sentence."
$E \to i$	"The first word must be an `i`"



More complicated example

Consider the following grammar. What does it mean?

$$B \to Bxy \mid Bz \mid q \mid r$$

- 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 $\begin{array}{l} E \to E+ \mid i \\ B \to Bxy \mid Bz \mid q \mid r \end{array}$
using the following transformation:

- Productions of the form $S \to \beta$ become $S \to \beta S'$.
- Productions of the form $S \to S\alpha$ become $S' \to \alpha S'$.
- Add $S' \to \epsilon$.

Result: $$\begin{array}{l} E \to iE' \\ E' \to +E' \mid \epsilon \\ B \to qB' \mid rB' \\ B' \to xyB' \mid zB' \mid \epsilon \end{array}$$

Mutual Recursions!

Things are slightly more complicated if we have mutual recursions.

$$\begin{array}{l} A \to Aa \mid Bb \mid Cc \mid q \\ B \to Ax \mid By \mid Cz \mid rA \\ C \to Ai \mid Bj \mid Ck \mid sB \end{array}$$

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.

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

First we eliminate the left recursion from A.

$A \rightarrow Aa \mid Bb \mid Cc \mid q$

becomes

$A \rightarrow BbA' \mid CcA' \mid qA'$
$A' \rightarrow aA' \mid \epsilon$

Left Recursion Example, 2

We substituting in the new definition of A, and now we will work on the B productions.

$A \rightarrow BbA' \mid CcA' \mid qA'$
$A' \rightarrow aA' \mid \epsilon$
$B \rightarrow Ax \mid By \mid Cz \mid rA$
$C \rightarrow Ai \mid Bj \mid Ck \mid sB$

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

$B \rightarrow Ax$ becomes
$B \rightarrow BbA'x \mid CcA'x \mid qA'x$

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...

$$C \rightarrow B\, bA'i \mid CcA'i \mid qA'i \mid B\, j \mid Ck \mid sB$$

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

$$\begin{aligned} C \rightarrow\ & CcA'xB'\, bA'i \mid qA'xB'\, bA'i \mid CzB'\, bA'i \mid rAB'\, bA'i \\ & CcA'xB'\, j \mid qA'xB'\, j \mid CzB'\, j \mid rAB'\, j \\ & CcA'i \mid qA'i \mid Ck \mid sB \end{aligned}$$

Left Recursion Example, 5

Reorganizing C, we have

$$\begin{aligned} C \to\ & qA'xB'bA'i \mid rAB'bA'i \mid qA'xB'j \mid rAB'j \mid qA'i \mid sB \\ & CcA'xB'bA'i \mid CzB'bA'i \mid CcA'xB'j \mid CzB'j \mid CcA'i \mid Ck \end{aligned}$$

Eliminating left recursion gives us

$$\begin{aligned} C \to\ & qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \\ & \mid rAB'jC' \mid qA'iC' \mid sBC' \\ C' \to\ & cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \\ & \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon \end{aligned}$$

The result...

Our final grammar is now

$$\begin{aligned} A &\to BbA' \mid CcA' \mid qA' \\ A' &\to aA' \mid \epsilon \\ B &\to CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \\ B' &\to bA'xB' \mid yB' \mid \epsilon \\ C &\to qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \\ &\quad \mid rAB'jC' \mid qA'iC' \mid sBC' \\ C' &\to cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \\ &\quad \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon \end{aligned}$$

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

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

Common Prefix

This grammar has common prefixes.

$$\begin{aligned} A &\to xyB \mid CyC \mid q \\ B &\to zC \mid zx \mid w \\ C &\to y \mid x \end{aligned}$$

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

Production	FirstSet
$A \to xyB$	$\{x\}$
$A \to CyC$	$\{x, y\}$
$A \to q$	$\{q\}$

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 $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$ we can rewrite it as $\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \end{aligned}$

So, in our example:

$$\begin{aligned} A &\to xyB \mid CyC \mid q \\ B &\to zC \mid zx \mid w \\ C &\to y \mid x \end{aligned} \quad \text{becomes} \quad \begin{aligned} A &\to xA' \mid q \mid yyC \\ A' &\to yB \mid yC \\ B &\to zB' \mid w \\ B' &\to C \mid x \\ C &\to y \mid x \end{aligned}$$

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