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.
- Know how to detect and eliminate conflicts with first and follow sets.

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

Example Input:  
+ 2 * 3 4

Syntax Tree:

Example Grammar:  
\[
S \rightarrow + EE \\
E \rightarrow \text{int} \\
E \rightarrow \ast EE
\]

Example Input:  
+ 2 * 3 4
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 → + E E
E → int
E → * E E
```

Example Input:

```
+ 2 * 3 4
```

Syntax Tree:

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

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 → + E E
E → int
E → * E E
```

Example Input:

```
+ 2 * 3 4
```

Syntax Tree:

```
S
  +
  |
  E
  |
  |
  2
  *
  |
  E
  |
  |
  3
  |
  |
  4
```
**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.

**Left Recursion**

**Left Recursion is Bad**

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

**Rules with Common Prefixes**

**Common Prefixes are Bad**

- A pair of rules rule like $E \to – E \mid – E E$ would confuse the function.
  Which version of the rule should be used?

**Things to Notice**

**Key Point — Prediction**

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

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

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

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

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

$E \rightarrow E +$  "We can have as many +s as we want at the end of the sentence."

$E \rightarrow i$  "The first word must be an i"

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 $x, y, z$
▶ At the beginning can come $q$ or $r$

Eliminating the Left Recursion

We can rewrite these grammars using the following transformation:

$E \rightarrow E + \mid i$

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

▶ 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 $S' \rightarrow \epsilon$.

Result:

$E \rightarrow iE'$

$E' \rightarrow +E' \mid \epsilon$

$B \rightarrow qB' \mid rB'$

$B' \rightarrow xyB' \mid zB' \mid \epsilon$

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.

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

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, 5

Reorganizing C, we have

\[ C \rightarrow qA'x'B'bA'i' | rAB'bA'i' | qA'x'B'j | rAB'j | qA'i' | sB \]
\[ CcA'x'B'bA'i' | CzB'bA'i' | CcA'x'B'j | CzB'j | CcA'i' \]

Eliminating left recursion gives us

\[ C \rightarrow qA'x'B'bA'i'C' | rAB'bA'i'C' | qA'x'B'jC' \]
\[ C \rightarrow CcA'x'B'bA'i'C' | CzB'bA'i'C' | CcA'x'B'jC' \]
\[ C \rightarrow CcA'x'B'bA'i'C' | CzB'bA'i'C' | CcA'x'B'jC' \]
\[ C \rightarrow CcA'x'B'bA'i'C' | CzB'bA'i'C' | CcA'x'B'jC' \]

The result...

Our final grammar is now

\[ A \rightarrow BbA' | CcA' | qA'i' \]
\[ A' \rightarrow aA' | \epsilon \]
\[ B \rightarrow CcA'x'B' | qA'x'B' | CzB' | rAB' \]
\[ B' \rightarrow bA'x'B' | yB' | \epsilon \]
\[ C \rightarrow qA'x'B'bA'i'C' | rAB'bA'i'C' | qA'x'B'jC' \]
\[ C \rightarrow CcA'x'B'bA'i'C' | CzB'bA'i'C' | CcA'x'B'jC' \]
\[ C \rightarrow CcA'x'B'bA'i'C' | CzB'bA'i'C' | CcA'x'B'jC' \]

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.

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

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

- **Production**
  - FirstSet
  - \( A \rightarrow xyB \) \{x\}
  - \( A \rightarrow CyC \) \{x, y\}
  - \( A \rightarrow q \) \{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 \( A \rightarrow \alpha_1 \beta_1 | \alpha_2 \beta_2 | \gamma \) we can rewrite it as \( A \rightarrow \alpha A' | \gamma \)

So, in our example:

- \( A \rightarrow xyB | CyC | q \) becomes \( A \rightarrow xa' | q | yyC \)
- \( A' \rightarrow yB | yC \)
- \( B \rightarrow zC | zx | w \) becomes \( B \rightarrow zB' | w \)
- \( B' \rightarrow C | x \)
- \( C \rightarrow y | x \)

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

- Epsilon productions have to be handled with care.

\[
A \rightarrow Bc \\
| x \\
B \rightarrow c \\
| \epsilon
\]

Is this LL?

- \(\text{FOLLOW}(B) = \{c\}\) and \(\text{FIRST}(B) = \{c\}\), so we have a conflict when trying to parse \(B\).
- We can substitute the \(B\) rule into the \(A\) rule to fix this...
- Be sure to check if you have introduced a common prefix though!

\[
A \rightarrow cc \\
| c \\
| x \\
⇒ \\
A \rightarrow cA' \\
| x \\
A' \rightarrow c \\
| \epsilon
\]