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: \[+ 2 \ast 3 4\]

Syntax Tree:

\[
\begin{array}{c}
S \\
E \\
E
\end{array}
\]

Derivations:

\[
S \\
+ E E
\]

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 F-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
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)

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:````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
**The Idea**

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

$$
\begin{align*}
E \rightarrow & \ E + \\
E \rightarrow & \ i
\end{align*}
$$

Derivation:

- $E$  
- $E-$  
- $E++$

---

**Rules with Common Prefixes**

A rule like $E \rightarrow -E \mid -EE$ would confuse the function. Which version of the rule should be used?
**Eliminating the Left Recursion**

We can rework these grammars using the following transformation:

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

**Result:**

$E \rightarrow iE'$

$E' \rightarrow E \mid \epsilon$

$B \rightarrow qB' \mid rB'$

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

---

**The Idea**

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

**Derivation:**

$E \rightarrow E +$

$E \rightarrow i$

---

**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 ($A$) 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:

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

$B \rightarrow Ax \mid By \mid Cz \mid rA$

$C \rightarrow Ai \mid Bj \mid Ck \mid sB$

First we eliminate the left recursion from $A$.

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

becomes

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

$A' \rightarrow aA' \mid \epsilon$

---

**More complicated example**

Consider the following grammar. What does $t$ 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 $r$
Left Recursion Example, 5

Reorganizing $C$, we have

$$C \rightarrow \begin{cases} qA'xBbA' \mid rAB'B \mid qA'x \mid rAB' \mid qA' \mid zB \\ Ca'C \mid CzBbA' \mid CaAk' \mid CzB'j \mid Ca'A' \mid Ck \end{cases}$$

Eliminating left recursion gives us

$$C \rightarrow \begin{cases} qA'x BbA'C' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rABj'C' \mid qA'iC' \mid sBC \\ Ca'C' \mid CzBbA'C' \mid CaC' \mid CaA'C' \mid Ck' \end{cases}$$

The result...

Our final grammar is now

$$A \rightarrow BbA' | CaA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow CcA'x | qA'x | BbA' | CzB' \mid rAB'B$$

$$B' \rightarrow bA'xB' \mid yB' \mid \epsilon$$

$$C \rightarrow qA'xB'ErA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rA'Bj'C' \mid qA'iC' \mid sBC$$

$$C' \rightarrow CA'xB'BA'iC' \mid zB'bA'iC' \mid CA'B'jC' \mid zB'jC' \mid CA'iC' \mid Ck' \mid \epsilon$$

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.

Left Recursion Example, 2

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

$$A \rightarrow BbA' | CaA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow A' \mid Bj \mid Cz \mid rA$$

$$C \rightarrow Ai \mid B'j \mid Ck \mid sB$$

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

$$B \rightarrow Ax \rightarrow BbA'x \mid CaA'x \mid qA'x$$

Left Recursion Example, 3

$$A \rightarrow BbA' | CaA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow BbA'x \mid CaA'x \mid qA'x \mid BbA' \mid Cz \mid rA$$

$$C \rightarrow Ai \mid Bj \mid Ck \mid sB$$

Now we can eliminate the simple left recursion in $B$, to get

$$B \rightarrow CcA'xB' \mid qA'x \mid BbA' \mid CzB' \mid rAB'B$$

$$B' \rightarrow bA'xB' \mid yB' \mid \epsilon$$

Left Recursion Example, 4

$$A \rightarrow BbA' | CaA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow CcA'xB' \mid qA'x \mid BbA' \mid CzB' \mid rAB'B$$

$$B' \rightarrow bA'xB' \mid yB' \mid \epsilon$$

$$C \rightarrow Ai \mid Bj \mid Ck \mid sB$$

Now production $C$: first, replace left recursive calls to $A$...

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

$$C \rightarrow \begin{cases} B • A'i | CcA'i | qA'i | B'j \mid Ck \mid sB \\ CaA'j | qA'j \mid Ck \mid sB \\ CcA' \mid qA' \mid Ck \mid sB \end{cases}$$

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.

Production | First Set
--- | ---
$A \rightarrow xyB$ | $\{x\}$

$A \rightarrow CyC$ | $\{x, y\}$

$A \rightarrow q$ | $\{q\}$

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?
Grammar 2

\[
S \rightarrow Ax \\
S \rightarrow By \\
A \rightarrow zB \\
B \rightarrow wA
\]

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

Left Factoring

If \( A = \alpha_1 \beta_1 \mid \alpha_2 \beta_2 \mid \gamma \) we can rewrite it as

\[
A \rightarrow A' \mid \gamma
\]

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

So, in our example:

\[
A = xyzB \mid C \gamma C \mid y q \quad \text{becomes} \quad A = zA' \mid y y C
\]

\[
B = zC \mid z x \mid w \\
A' = y B \mid y C
\]

\[
C = y x \\
E = z B' \mid w \\
E' = C \mid x \\
C = y x
\]

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

Grammar 3

\[
S \rightarrow Ax \\
S \rightarrow By \\
A \rightarrow zB \\
B \rightarrow wA
\]

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.

\[
S \rightarrow z S' \\
S' \rightarrow z x \\
S' \rightarrow y
\]

Activity

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

Grammar 1

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

Grammar 2

\[
E \rightarrow A \, x \\
E \rightarrow B \, y \\
A \rightarrow z \, B \\
B \rightarrow w \, A
\]

Grammar 3

\[
E \rightarrow A \, x \\
E \rightarrow B \, y \\
A \rightarrow z \, B \\
B \rightarrow w \, A
\]

Lambda-Calculus Concrete Syntax

```
<expression> -->
  <identifier> |
  (lambda (<identifier>) <expression>) |
  (<expression> <expression>)
```

Grammar 1

Grammar 1 starts as: Eliminate the left-recursion to get: Fix the common prefixes to get:

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

\[
E' \rightarrow E'' \\
E' \rightarrow y \, E' \\
E' \rightarrow B \, E' \\
B \rightarrow z
\]

\[
E \rightarrow x \, E'' \\
E \rightarrow y \, E' \\
E \rightarrow E' \\
B \rightarrow E \, z
\]
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?)))

Parsing Lambda-Calculus Expressions

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

Unparsing – Wh

{(define unpars (lambda exp (cases expr (var-exp (lambda-ex (list 'l unpars (app-exp (list ())))

University of Illinois