Announcements

- May redo 1st HW and resubmit by Monday.
- 2nd HW due in 1 week.
- Midterm - Monday before spring break

(ambiguous CNF)
Other parsing algorithms

CYK is still pretty slow, especially for large programming languages. \(O(n^3)\)

After it was developed, a lot of work was put into figuring out what grammars could have faster algorithms.

Two big (and useful) classes have \(O(n)\) time parsers: LL and LR.
LL & LR grammars

"LL" is left-to-right, leftmost derivation

"LR" is left-to-right, rightmost derivation

* So parser will scan left to right either way.

* LL will make a leftmost derivation (so right-leaning tree)

  ( )
LL versus LR

- LL are a bit simpler, so we'll start with them.

- Note: LR is a larger class (so more grammars are LR than are LL).

- Both are used in production compilers today.
Example: LL parsing

\[
\begin{align*}
\text{idlist} & \rightarrow \text{id} \quad \text{idlist\_tail} \\
\text{idlist\_tail} & \rightarrow \quad \text{id} \quad \text{idlist\_tail} \\
\text{idlist\_tail} & \rightarrow \quad \cdot \\
\text{idlist\_tail} & \rightarrow \quad \cdot \\
\end{align*}
\]

Parse tree for A, B, C:

```
  idlist
   /  \
  idlist
    /  \n  id(A)  idlist\_tail
       /  \
      idlist\_tail
         /  \n        id(B)  idlist\_tail
           /  \
          idlist\_tail
            /  \n           id(C)
```
**LL(1), LL(2)**

**LL(k) + LR(k)**

When LL or LR is written with (1), (2), etc., it refers to how much look-ahead is allowed.

LL(1) means we can only look 1 token ahead when making our decision of which rule to match.

Most commercial ones are LR(i), but exceptions exist, such as ANTLR.
A non LL(1) example: Left recursion

\[
\text{id-list} \rightarrow \text{id} \hspace{1cm} \text{id-list}, \text{id} \quad \text{LR}
\]

Imagine: Scanning left to right, if you encounter an id token, which parse tree do we build?

A, B, C

\[
\begin{array}{c}
\text{id-list} \\
\text{id-list}, \text{id} \\
\text{id-list} \\
\text{id-list}, \text{id} \\
\text{id-list} \\
\text{id-list}, \text{id} \\
\text{id-list} \rightarrow \text{id} \\
A, B, C
\end{array}
\]
Making the grammar LL(1):

\[ \text{id-list} \rightarrow \text{id} \quad \text{id-list-tail} \]

\[ \text{id-list-tail} \rightarrow \text{id} \quad \text{id-list-tails} \quad \epsilon \]

\[ \overrightarrow{A, B, C} \]

\[ \overrightarrow{\text{id-list}} \]

\[ \overrightarrow{\text{id(A)}} \quad \overrightarrow{\text{id-list-tail}} \]

\[ \overrightarrow{\text{id(B)}} \quad \overrightarrow{\epsilon} \]

\[ \overrightarrow{\epsilon} \]
Another non-LL(0) example: common prefixes

\[
\text{stmt} \rightarrow \begin{cases} 
  \text{id} \rightarrow \text{expr} \\
  \text{id} \rightarrow (\text{argument-list}) 
\end{cases}
\]

So when next token is an id, don't know which rule to use.

Fix?

\[
\text{stmt} \rightarrow \text{id} \text{ stmt-tail} \quad \text{A} := \text{B + C} 
\]

LL(0)

\[
\text{stmt-tail} \rightarrow := \text{expr} \\
\rightarrow (\text{argument-list}) \\
\text{id}(\text{stmt-tail}) 
\]

\[\rightarrow \text{ABC} \]
Some grammars are non-LL:

- Eliminating left recursion and common prefixes is a very mechanical procedure which can be applied to any grammar.

- However, might not work! There are examples of inherently non-LL grammars.

- In these cases, generally add some heuristic to deal with odd cases
Example: non-LL language

\[ \text{stmt} \rightarrow \mathcal{R} \text{ condition then-clause else-clause} \]

\[ \text{then-clause} \rightarrow \text{then } \text{stmt} \]
\[ \text{else-clause} \rightarrow \text{else } \text{stmt} \]

What syntax?

```
if __________
then __________
else __________
```
Ex: if $C_1$ then if $C_2$ then $S_1$ else $S_2$

Parse tree:

```
  stmt
    /
  if
    /
    /
  if
    /
    /
  if
    /
    /
  $C_1$
  thenblock
    /
    /
  if
    /
    /
  if
    /
    /
  $C_2$
  thenblock
```

No possible grammar for if statements
Back to LL-parsing

We have seen mostly top-down parsing.

Start with So, the start token, and try to construct the tree based on the next input.

Bottom-up Parsing starts at the leaves (here, the tokens) and tries to build the tree upward.

Continues scanning & shifting tokens onto a forest, then builds up when it finds a valid production.
Bottom-up parsing

idlist → id idlist\_tail

idlist\_tail → , id idlist\_tail

idlist\_tail → ·

Ex: A, B, C

Bottom-up parsing:

(is this left-most or right-most?)

leftmost derivation
Shift-reduce:
- Bottom up parsers are also called shift-reduce:
  - Shift token onto stack
  - When a rule is recognized, reduce to left-hand side

- Problem with last example: must shift all tokens onto the stack before reducing. What could happen in a large program? Overflow your stack?

- Sometimes unavoidable. However, sometimes other options...
Bottom-up parsing: another example

\[
\begin{align*}
\text{id-list} & \rightarrow \text{id-list-prefix} \cdot \text{id} \\
\text{id-list-prefix} & \rightarrow \text{id-list-prefix} \cdot \text{id} \\
\text{id} & \rightarrow \text{id}
\end{align*}
\]

Parse \( A, B, C \) again, bottom-up:

\[
\begin{align*}
\text{id-list} & \rightarrow \text{id-list-prefix} \cdot \text{id(C)} \\
\text{id-list-prefix} & \rightarrow \text{id-list-prefix} \cdot \text{id(B)} \\
\text{id} & \rightarrow \text{id(A)}
\end{align*}
\]
Bottom-up parsing: some notes

- The previous example cannot be parsed top-down. (left recursion)

- Note that it also is not an LL grammar, although the language is LL.

- There is a distinction between a language and a grammar. Remember, any language can be generated by an infinite number of grammars.