Announcements

- HW due Saturday
Other parsing algorithms

CYK is still pretty slow, especially for large programs.

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 and 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

\[ \text{idlist} \rightarrow \text{id} \ \text{idlist\_tail} \]

\[ \text{idlist\_tail} \rightarrow \text{id} \ \text{idlist\_tail} \]

\[ \text{idlist\_tail} \rightarrow \text{\_j} \]

Parse tree for "A, B, C, j" → id(A), id(B), id(C), j
id_list

id_list

id(A)  id_list_tail

id_list

id(A)  id_list_tail

id(B)  id_list_tail

id_list

id(A)  id_list_tail

id(B)  id_list_tail

id(C)  id_list_tail

;
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(1) but exceptions exist (such as ANTLR).
A non LL(1) example: Left recursion

\[
\text{idlist} \rightarrow \text{id} \\
\rightarrow \text{id-list, id}
\]

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

\[ A, B, C \]
Making the grammar LL(1):

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

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

\[ \rightarrow \varepsilon \]

A, B, C
Another non-LL(0) example: common prefixes

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

So when next token is an \text{id},

don't know which rule to use.

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

\[
c = x + y \]
\[
c(x) \]

\text{PASCAL}
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: optional else

\[
\text{stmt} \rightarrow \text{if } \text{condition } \text{then-clause } \text{else-clause} \\
\text{then-clause} \rightarrow \text{then } \text{stmt} \\
\text{else-clause} \rightarrow \text{else } \text{stmt} \\
\text{what syntax?} \rightarrow \epsilon
\]

PASCAL if statement
Ex: if $C_1$ then if $C_2$ then $S$ else $S_2$

Parse tree:

Inherently ambiguous
Back to LL-parsing

We have seen mostly top-down parsing.

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

Also called predictive parsing matches the rule based on current token/state plus the next input or next k inputs
LR grammars

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

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

Never predicts - when it recognizes right hand side of a rule, simplifies to left hand side.
Bottom-up parsing

\[
\text{idlist} \rightarrow \text{id} \ \text{idlist}_{-tail}
\]

\[
\text{idlist}_{-tail} \rightarrow \text{id} \ \text{idlist}_{-tail}
\]

\[
\text{idlist}_{-tail} \rightarrow \ldots
\]

Ex.

\[
\text{idlist} \rightarrow \text{id} \ \text{idlist}_{-tail} \\
\text{idlist}_{-tail} \rightarrow \text{id} \ \text{idlist}_{-tail} \\
\text{idlist}_{-tail} \rightarrow \ldots
\]

\[
\text{id(A)}, \ \text{id(B)}, \ \text{id(C)}
\]
id(A), id(B)
id(A), id(B),
id(A), id(B), id(C)
id(A), id(B), id(C); id_list_tail
id(A), id(B) id_list_tail
  , id(C) id_list_tail
    ;

id(A) id_list_tail
  , id(B) id_list_tail
    , id(C) id_list_tail
      ;

id_list
  id(A) id_list_tail
    , id(B) id_list_tail
      , id(C) id_list_tail
        ;
**Shift-reduce**:

- Bottom-up parsers are also called **shift-reduce**:
  - Shift token onto stack (in a forest)
  - When a rule is recognized, reduce to left-hand side

Problem with last example:
- Must shift all tokens onto the forest before reducing.

What could happen in a large program?
- Slow, might crash/overflow

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

\[ \text{id-list} \rightarrow \text{id-list-prefix} \]

\[ \text{id-list-prefix} \rightarrow \text{id-list-prefix} \circ \text{id} \]

Parse A, B, C; again, bottom-up:
**Bottom-up parsing: some notes**

- The previous example cannot be parsed top-down. Why? *not LL: 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.
LR grammars: An old example

```
expr → term | expr add-op term
term → factor | term multop factor
factor → id | number | - factor | (expr)
add-op → + | -
multop → * | /
```
This grammar is not LL!

- If we get an id as input when expecting an expr, no way to choose between the 2 possible productions.

- It suffers from the common prefix issue we saw before.

(We can fix this.)
Another LL example:

\[
\begin{align*}
\text{expr} & \rightarrow \text{term} \text{ term}_{-\text{tail}} \\
\text{term}_{-\text{tail}} & \rightarrow \text{add}_{-\text{op}} \text{ term} \text{ term}_{-\text{tail}} \\
& \rightarrow \varepsilon \\
\text{term} & \rightarrow \text{factor} \text{ factor}_{-\text{tail}} \\
\text{factor}_{-\text{tail}} & \rightarrow \text{mult}_{-\text{op}} \text{ factor} \text{ factor}_{-\text{tail}} \\
& \rightarrow \varepsilon \\
\text{factor} & \rightarrow (\text{expr}) \mid \text{id} \mid \text{number} \\
\text{add}_{-\text{op}} & \rightarrow + \mid - \\
\text{mult}_{-\text{op}} & \rightarrow * \mid / 
\end{align*}
\]
Now can add this as part of a simple calculator language:

```
program → stmt_list $$
stmt_list → Stmt stmt_list
          → ε
stmt → id := expr
      → read id
      → write expr
```

end of file
Program: What does it do?

read A

read B

sum := A + B

write sum

write sum / 2

How to parse?