Announcements

- Office hours today - 3-4

- For HW3, not just wanting pattern matching description

  gcc file lex -lfl

  (man gcc)

  HW3 due Monday by 11:59 pm
LL parsing
Left-to-right, leftmost derivation.

Usually top-down.

Things that prevent it:
- left recursion
- common prefixes

But not all languages are LL!
Example: LL parsing

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

Parse tree for \text{A, B, C, j}

[Diagram of a parse tree for A, B, C, j is shown on the page.]
Bottom-up parsing: another example

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

Left recursion

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

\[
\text{id\text{-}list} \rightarrow \text{id\text{-}list\text{-}prefix} \ \vdash \ \text{id\text{-}list\text{-}prefix} \rightarrow \text{id\text{-}list\text{-}prefix} \ \vdash \ \text{id\text{-}list\text{-}prefix} \ \vdash \ \text{id}(C) \ \vdash \ \text{id\text{-}list\text{-}prefix} \ \vdash \ \text{id}(B) \ \vdash \ \text{id}(A)
\]

Rightmost derivation
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.
LR grammars: An old example

\[
\begin{align*}
\text{expr} & \rightarrow \text{term} \mid \text{expr add-op term} \\
\text{term} & \rightarrow \text{factor} \mid \text{term multop factor} \\
\text{factor} & \rightarrow \text{id} \mid \text{number} \mid - \text{factor} \mid (\text{expr}) \\
\text{add-op} & \rightarrow + \mid - \\
\text{multop} & \rightarrow * \mid / \\
\end{align*}
\]

(What does LR stand for?)

Rightmost derivation (so left tree)
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.

(Also left recursion)

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

(See sec. 2.3.1 for an example LL grammar that is equivalent.)
Building an \( LL \) parsers

2 options:

1. Recursive descent parser: code whose subroutines correspond to non-terminals (like case statement version of scanner)

2. \( LL \) parse table which is used by a driver program (like bison). (like flex)
Comparing the 2 options

- Both recursive descent and parse tables are used in practice.
- Generally, recursive descent code is handwritten for smaller languages (or when tool is unavailable).
- Exceptions exist, however. Example: gcc
Neither U or LR:

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

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

This one is inherently ambiguous.

Solution? add rule to force uniformity
Another way: avoid ambiguity
Use an explicit end:

\[
\text{stmt} \rightarrow \text{IF condition then-clause else-clause END} \\
\quad \text{other-stmt}
\]

then-clause \rightarrow \text{then } \text{stmt-list}

else-clause \rightarrow \text{else } \text{stmt-list} \quad | \\
\quad (\text{Python - indent})
Bottom-up parsing

- Uses a stack to push tokens onto.

Why?

Constant time!
Good use of space

Returns tokens in order I want
LR version of calculator

```
program → stmt-list $$
stmt-list → stmt-list stmt | stmt
stmt → id := expr | read ID | write expr
expr → term | expr add-op term
term → factor | term mult-op factor
factor → (expr) | id | number
add-op → + | -
mult-op → * | /
```
Example to parse:
read A
read B
sum := A + B
write sum
write sum/2

Beginning: know we start with program.

So candidates are:
program -> stmt-list $$

what else?
(fake closure)
Big picture: LR parsing
- Keep track of states it has traversed by pushing them into the parse stack along with symbols.
- When top of stack (input + states) says we need to reduce using a rule $A \rightarrow \alpha$:
  - pop len($\alpha$) items off stack
  - push $A$ on stack
LR: Main issue
- Need to deal with conflicts.
  Might have 2 possible rules we match, one of which calls for a shift (or push onto stack) or the other a reduce (A → a).
- An LR(0) parser works only when no such conflicts exist.
- Any language which can be parsed bottom-up has an LR(0) grammar—but not practical.
Other LR strategies

- SLR (or simple LR) parsers look at future inputs. Will only reduce $A \rightarrow \alpha$ if the next $\gamma$ tokens could follow $\alpha$ in the grammar.

- LALR (look ahead LR) parsers improve on SLR by using local, state-specific look ahead.

- LALR is most common in practice.
Syntax Errors

When parsing a program, will often detect syntax errors where tokens do not form valid states.

What should we do?

Find closest rule that does match.

"Recover" — continue parsing.
Generating good error messages

Most compilers do not just halt; this would mean code past the first error is ignored. However, it is beneficial to detect as many errors as possible.

So how to continue after an error?
Approaches

1. Panic mode: define a small set of "safe symbols".
   - In C++, use semicolon
   - In Python: /n

    When error occurs, compiler deletes back to the last safe symbol

    (Ever notice that errors often point to the line before or after the actual error?)
2) Phase-level recovery:
refinement of panic mode with
different safe symbols in different
states.

Ex: expression → )
    statement → ;

3) Context specific look-ahead:
   improves on 2) by checking
   various contexts in which
   the production might appear
   in the parse tree.
Beyond Parsing (Ch. 4)

Need rules to connect the productions to actual concepts.

Ex:

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T \ast F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow - F \\
F & \rightarrow (E) \\
F & \rightarrow \text{const}
\end{align*}
\]

(LR(0) or LL(0) or neither?)

not LL - left recursion!
This grammar generates all well-formed constant expressions.

However, says nothing about their meaning.

Need to tie these to basic rules for the processor.

These can be specified in machine language or some other existing code on the machine.
Example: Associate a value with each nonterminal (we assume const is given by scanner).

\[
E \rightarrow E + T : \quad E_1 \rightarrow E_2 + T
\]

\[
E_1 \cdot \text{val} := \text{sum} \ (E_2 \cdot \text{val}, T \cdot \text{val})
\]

\[
E \rightarrow T : \quad E \cdot \text{val} := T \cdot \text{val}
\]

\[
E \rightarrow \text{const} : \quad F \cdot \text{val} := \text{const} \cdot \text{val}
\]