Objectives: today
- Issues in designing a grammar
- AST extensions for the 401 project
- Overview of parsing algorithms
- Motivation and details of top-down, predictive parsers
- Recursive descent parsing
- Today++: a walk through the PL/0 parser

Next week
- Monday: no class
- Wednesday: Mark S. on bottom-up parsing
- Friday: TBA
  - Maybe a lecture
  - Maybe a tools scavenger hunt
  - Maybe ...

Designing a grammar: on what basis?
- Accuracy
- Readability, clarity
- Unambiguity
- Limitations of CFGs
- Ability to be parsed by a particular parsing algorithm
  - Top-down parser => LL(k) grammar
  - Bottom-up parser => LR(k) grammar

AST extensions in project (I)
- Expressions
  - true and false constants
  - array index expression (an lvalue)
  - function call expression
  - and and or operations
  - tests are expressions
  - constant expressions
- Statements
  - for
  - break
  - return
  - if with else
- Declarations
  - procedures with result types
  - var parameters
- Types
  - bool
  - array

Parsing algorithms
- Given input (sequence of tokens) and grammar, how do we find an AST that represents the structure of the input with respect to that grammar?
- Two basic kinds of algorithms
  - Top-down: expand from grammar’s start symbol until a legal program is produced
  - Bottom-up: create sub-trees that are merged into larger sub-trees, finally leading to the start symbol
Top-down parsing

- Build AST from top (start symbol) to leaves (terminals)
- Represents a leftmost derivation (e.g., always expand leftmost non-terminal)
- Basic issue: when replacing a non-terminal with a right-hand side (rhs), which rhs should you use?
- Basic solution: "Use the input tokens, Luke!"

Predictive parser

- A top-down parser that can select the correct rhs looking at at most k input tokens (the lookahead)
- Efficient
  - No backtracking is needed
  - Linear time to parse
- Implementation
  - Table-driven: pushdown automaton (PDA) — like table-driven FSA plus stack for recursive FSA calls
  - Recursive-descent parser [used in PL/0]
    - Each non-terminal parsed by a procedure
    - Call other procedures to parse sub-non-terminals, recursively

LL(k), LR(k), …?

- These parsers have generally snazzy names
- The simpler ones look like the ones in the title of this slide
  - The first L means "walk the token sequence left to right"
  - The second letter means "produce a (right)leftmost derivation"
    - Leftmost ⇒ top-down
    - Rightmost ⇒ bottom-up
  - The k means "k tokens of lookahead"
- We won’t discuss LALR(k), SLR, and lots more parsing algorithms

LL(k) grammars

- It’s easy to construct a predictive parser if a grammar is LL(k)
  - Left-to-right scan on input, Leftmost derivation, k tokens of lookahead
  - Restrictions include
    - Unambiguous
    - No common prefixes of length ≥ k
    - No left recursion
  - Collectively, the restrictions guarantee that, given k input tokens, one can always select the correct rhs to expand

Eliminating common prefixes

- Left factor them, creating new non-terminal for the common prefix and/or different suffixes
- Before
  - If ::= if Test then Stmts end | if Test then Stmts else Stmts end
- After
  - If ::= if Test then Stmts IfCont
    - IfCont ::= and | else Stmts end
  - Grammar is a big uglier
  - Easy to do manually in a recursive-descent parser

Eliminating left recursion: rewrite the grammar

- Before
  - E ::= E + T | T
  - T ::= T * F | F
  - F ::= id | ( E ) | ...
- After
  - E ::= T ECont
    - ECont ::= + T ECont | ε
  - T ::= F TCont
    - TCont ::= * F TCont | ε
  - F ::= id | ( E ) | ε
Just add sugar

- \( E ::= T \{ + T \} \)
- \( T ::= F \{ * F \} \)
- \( F ::= id \mid ( E ) \mid ... \)

- Sugared form is still pretty readable
- Concrete syntax tree is not as close to abstract syntax tree
- Easy to implement in hand-written recursive descent parser

Table-driven predictive parser

- Automatically compute PREDICT table from grammar
- \( \text{PREDICT}(\text{nonterminal}, \text{input-symbol}) \Rightarrow \text{rhs} \)

Example

\[
\begin{align*}
\text{Stmt} & ::= 1 \text{ if } \text{expr} \text{ then } \text{Stmt} \text{ else } \text{Stmt} \mid \\
& 2 \text{ while } \text{Expr} \text{ do } \text{Stmt} \mid \\
& 3 \text{ begin } \text{Stmts} \text{ end} \mid \\
\text{Stmts} & ::= 4 \text{ Stmt } ; \text{Stmts} \mid 5 \varepsilon \\
\text{Expr} & ::= 6 \text{ id} \\
\end{align*}
\]

Construct PREDICT table

- Compute FIRST set for each rhs
  - All tokens that can appear first in a derivation from that rhs
- In case rhs can be empty
  - Compute FOLLOW set for each non-terminal
    - All tokens that can appear right after that non-terminal in a derivation
- Compute FIRST and FOLLOW sets mutually recursively
- PREDICT then depends on the FIRST set

Another example

\[
\begin{align*}
\text{E} & ::= T \{ + T \} \\
\text{T} & ::= F \{ * F \} \\
\text{F} & ::= - F \mid \text{id} \mid ( E ) \\
\text{E}' & ::= 1 \text{ T E}' \\
\text{E''} & ::= 2 + T \text{ E''} \mid 3 \varepsilon \\
\text{T} & ::= 4 \text{ F T}' \\
\text{T'} & ::= 5 * \text{ F T'} \mid 6 \varepsilon \\
\text{F} & ::= 7 - F \mid 8 \text{id} \mid 9 ( E ) \\
\end{align*}
\]
PREDICT

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>(</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PREDICT and LL(1)

- If PREDICT table has at most one entry per cell
  - Then the grammar is LL(1)
  - There is always exactly one right choice
    - So it's fast to parse and easy to implement
  - LL(a) ↔ each column labeled by one token
- Can have multiple entries in each cell
  - Ex: common prefixes, left recursion, ambiguity
  - Can patch table manually, if you “know” what to do
  - Or can use more powerful parsing technique