Objectives: today

- Recap and clarify PREDICT table
- Describe computation of FIRST and FOLLOW
  - And the relationship to PREDICT
- Recursive descent parsing
  - High-level issues and
  - (time-permitting) a walk through the PL/0 parser

Next week

- Monday: no class
- Wednesday: Mark Seigle
- Friday: Matthai Philipose

Example

Stmt ::= 1 if Expr then Stmt else Stmt | 2 while Expr do Stmt | 3 begin Stmts end
Stmts ::= 4 Stmt ; Stmts | 5 ε
Expr ::= 6 id

<table>
<thead>
<tr>
<th>if</th>
<th>then</th>
<th>else</th>
<th>while</th>
<th>do</th>
<th>begin</th>
<th>end</th>
<th>id</th>
<th>;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Compute PREDICT

In terms of FIRST and FOLLOW

- FIRST(α)
  - $α$ is any string of tokens
  - FIRST(α) is the set of terminals that being strings derived from $α$
  - If $α$ can derive $ε$, then include $ε$ in FIRST
- FOLLOW(A)
  - $A$ is a non-terminal
  - FOLLOW(A) is the set of terminals that can appear immediately after $A$ in some derivation in the language
  - If $A$ can be the end of such a derivation, then include $ε$ in FOLLOW
  - $ε$ represents “end of input”
Computing \textsc{FIRST}(X)

for all grammar symbols \(X\)

- If \(X\) is a terminal, then \textsc{FIRST}(X) is \([X]\)
- If \(X ::= \varepsilon\), then add \(\varepsilon\) to \textsc{FIRST}(X)
- If \(X\) is a non-terminal and \(X ::= Y_1 Y_2 \ldots Y_k\)
  - \textsc{FIRST}(X) is initialized to \textsc{FIRST}(Y_1)
  - If \(\varepsilon \in \textsc{FIRST}(Y_1)\) then union in \textsc{FIRST}(Y_2)
  - If \(\varepsilon \in \textsc{FIRST}(Y_1)\) and \(\varepsilon \in \textsc{FIRST}(Y_2)\) then union in \textsc{FIRST}(Y_3)
  - Etc.
  - If \(\varepsilon \in \textsc{FIRST}(Y_i)\) for all \(i\), then add \(\varepsilon\) to \textsc{FIRST}(X)

Computing \textsc{FIRST}(X_1 X_2 \ldots X_n)

- Start with all the non-\(\varepsilon\) symbols in \textsc{FIRST}(X_1)
- If \(\varepsilon \in \textsc{FIRST}(X_1)\), then union in all the non-\(\varepsilon\) symbols in \textsc{FIRST}(X_2)
- If \(\varepsilon \in \textsc{FIRST}(X_1)\) and \(\varepsilon \in \textsc{FIRST}(X_2)\), then union in all the non-\(\varepsilon\) symbols in \textsc{FIRST}(X_3)
- Etc.
- If \(\varepsilon \in \textsc{FIRST}(X_i)\) for all \(i\), then add \(\varepsilon\) to \textsc{FIRST}(X_1 X_2 \ldots X_n)

Computing \textsc{FOLLOW}(A)

for all non-terminals \(A\)

- Place \$ in \textsc{FOLLOW}(S), where \(S\) is the start symbol and \$ is the end-of-input marker
- If there is a production \(A ::= \alpha B\beta\) then everything in \textsc{FIRST}(\beta) except \(\varepsilon\) is placed in \textsc{FOLLOW}(B)
- If there is a production \(A ::= \alpha B\) (or \(A ::= \alpha B\beta\) where \(\varepsilon \in \textsc{FIRST}(\beta)\)) then everything in \textsc{FOLLOW}(A) is placed in \textsc{FOLLOW}(B)
- Apply until a fixpoint is reached

Another example

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

Construct predictive parsing table

- Input is a grammar \(G\)
- Output is a table \(M\) indexed by non-terminals (rows) and terminals (columns)
- foreach production \(A ::= \alpha\) in \(G\) do
  - foreach terminal \(a\) in \textsc{FIRST}(\alpha) add the production to \(M[A,a]\);
  - if \(\varepsilon \in \textsc{FIRST}(\alpha)\) add the production to \(M[A,\$]\) for each terminal \(b\) in \textsc{FOLLOW}(A);
  - if \(\varepsilon \in \textsc{FIRST}(\alpha)\) and \(\$ \in \textsc{FOLLOW}(A)\) add the production to \(M[A,\$]\)
- Every remaining undefined entry is an error
PREDICT and LL(1)

- If the PREDICT table has at most one entry in each cell, then the grammar is LL(1)
  - And there is only one choice (it’s predictive), making it fast to parse and easy to implement
- Multiple entries in a cell
  - Arise with left recursion, ambiguity, common prefixes, etc.
  - Can patch by hand
  - Or use more powerful parsing techniques

Recursive descent parsers

- Write procedure for each non-terminal
- Each procedure selects the correct right-hand side by peeking at the input tokens
- Then the r.h.s. is consumed
  - If it’s a terminal symbol, verify it is next and then advance through the token stream
  - If it’s a non-terminal, call corresponding procedure
- Construct and return AST representing the r.h.s.

It’s demo time…

- Let’s look at some of the PL/0 code to see how the recursive descent parsing works in practice