Predictive Parsing (finish) & Bottom-Up Parsing

CS2210
Lecture 5

Non-recursive Predictive Parsers

- Avoid recursion for efficiency reasons
- Typically built automatically by tools

```
Input  a  b  $  
Stack  X  Y  $  
```

```
Predictive Parsing Program
M[ , ] gives production A symbol on stack a input symbol (and $)

Stack contents and remaining input called parser configuration (initially $S on stack and complete input string)
```

1. If X=a=halt and announce success
2. If X=a $ pop X off stack advance input to next symbol
3. If X is a nonterminal use M[X,a] which contains production X->rhs or error replace X on stack with rhs or call error routine, respectively, e.g. X-U>V=W replace X with VWU (U on top) output the production (or augment parse tree)
# Construction of Parsing Table Helpers (1)

- **First(α)** = set of terminals that begin strings derived from α
  - First(X) = {X} for terminal X
  - If X → ε a production add ε to First(X)
  - For X → Y₁...Yₖ place a in First(X) if a in First(Yᵢ) and ε ∈ First(Yᵢ) for j = 1...i-1, if ε ∈ First(Yᵢ) j = 1...k add ε to First(X)

# Construction of Parsing Table Helpers (2)

- **Follow(A)** = set of terminals a that can appear immediately to the right of A in some sentential form i.e., S →* α Aβ for some α, β (α can include $)
  - Place $ in Follow(S), S start symbol, $ right end marker
  - If there is a production A → αβ put everything in First(β) except ε in Follow(β)
  - If there is a production A → αB or A → αβ where ε is in First(β) then everything in Follow(A) is in Follow(B)

# Construction Algorithm

Input: Grammar G  
Output: Parsing table M

For each production A → α do  
  For each terminal a in FIRST(α) add A → α to M[A, a]  
  If ε is in FIRST(α) add A → α to M[A, b] for each terminal b in FOLLOW(A).  
  ($ counts as a terminal in this step)  
Make each undefined entry in M to error
Example

\[
E \rightarrow TE' \\
E' \rightarrow +TE' | x \\
T \rightarrow FT' \\
T' \rightarrow +FT' | x \\
F \rightarrow (E) | id
\]

\[
\begin{array}{|c|c|c|c|}
\hline
id & + & ( & $ \\
\hline
E & & & \\
E' & & & \\
T & & & \\
T' & & & \\
F & & & \\
\hline
\end{array}
\]

**FIRST**:
- \( \text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{, , id \} \)
- \( \text{FIRST}(E') = \{ , , +, \epsilon \} \)
- \( \text{FIRST}(T') = \{ , , \ast, \epsilon \} \)

**FOLLOW**:
- \( \text{FOLLOW}(E) = \text{FOLLOW}(E') = \{ , , ) , , , \} \)
- \( \text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ , , +, \ast, \epsilon \} \)
- \( \text{FOLLOW}(F) = \{ , , +, \ast, \epsilon \} \)

---

**LL(1)**

- A grammar whose parsing table has no multiply defined entries is said to be **LL(1)**
- First \( L \) = left to right input scanning
- Second \( L \) = leftmost derivation
- \( (1) \) = 1 token lookahead
- Not all grammars can be brought to **LL(1)** form, i.e., there are languages that do not fall into the **LL(1)** class

---

**Bottom Up Parsing**
Shift-Reduce Parsing

- Two actions
  - Reduce -> replace an input substring that matches the rhs of a production with the lhs nonterminal
  - Shift -> read one more input token
- Two forms
  - Operator precedence parsing
  - LR parsing
    - Left-to-right scanning
    - Rightmost derivation

Implementation with a stack

<table>
<thead>
<tr>
<th>Grammar:</th>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>E -&gt; E + E</td>
<td>$</td>
<td>id1 + id2 * id3</td>
<td>shift</td>
</tr>
<tr>
<td>E -&gt; E * E</td>
<td>E</td>
<td>id1 * id2</td>
<td>shift</td>
</tr>
<tr>
<td>E -&gt; (E)</td>
<td>E</td>
<td>id1 E</td>
<td>shift</td>
</tr>
<tr>
<td>E -&gt; id</td>
<td>$</td>
<td>id</td>
<td>accept</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar:</th>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$</td>
<td>id1 + id2 * id3</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>+ id2 * id3</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>+ id2 * id3</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>+ id2 * id3</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>id1</td>
<td>* id2</td>
<td>reduce: E -&gt; id</td>
</tr>
</tbody>
</table>

Shift-reduce Conflicts

- stmt -> if expr then stmt | if expr then stmt else stmt | other
- Stack: ... if expr then stmt
- Input: else ...$
  - Don't know whether to reduce or shift
  - Grammar not LR(1) (ambiguous)
  - Change grammar or resolve by shifting (tools such as yacc do this)
LR Parsing

- Can be constructed for almost all programming languages constructs
- LR most general non-backtracking shift-reduce method and efficiently implementable
- LR ⊃ LL, i.e., can parse more grammars than predictive parsers
- Detects syntax errors as early as possible in a left-to-right scan
- But: difficult to construct by hand

LR Parsers

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>π₀ → X₀ → π₁ → X₁ → π₂ → … → Xₘ → s₀, a₀, a₁, …, aₙ</td>
<td>a₁ → a₀ → aₙ → $</td>
</tr>
</tbody>
</table>

.action[aₙ, a₀] =
1. Shift a
2. Reduce A → β
3. Accept
4. Error

Actions

- Configuration:
  - (s₀ X₁ s₂ X₃ s₄ … Xₘ s₀ a₁, a₂, …, aₙ)$
- shift s
  - (s₀ X₁ s₂ X₃ s₄ … Xₘ s₀ a₁, a₂, …, aₙ)$
- reduce A → β (r = |β|)
  - (s₀ X₁ s₂ X₃ s₄ … Xₘ s₀, Xₘ, A r s₁, aₙ, …, aₙ)$
    - $ = \text{goto}[sₐₙ₋₁A]$
    - $Xₘ₋₁ … X₀ = β$
- accept
- error
Parsing Example

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input: id*id+id$

Stack: 0

LR Grammars

- a grammar for which a LR parsing table can be constructed
- LR(0) and LR(1) typically of interest
  - What about LL(0)?
- Parsing tables from LR grammars
  - SLR (simple LR) tables
    - Many grammars for which it is not possible
  - Canonical LR tables
    - Works on "most" grammars
  - LALR (= lookahead LR) tables
    - Often used in practice because significantly smaller than canonical
LR(0) items
- A->XYZ
- Items:
  - A->.XYZ
  - A->X.YZ
  - A->XY.Z
  - A->XYZ.
- Represents how much of a production we have seen in parsing

Parse Table Construction
- Build a DFA to recognize viable prefixes
  - set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser
  - Items are states of an NFA for viable prefixes
    - Group items together (subset construction) to get DFA
    - Define a canonical collection of LR(0) items
- Helpers:
  - Augmented grammar
    - S start symbol of G, S'->S new start symbol S'
  - Closure function
  - Goto function

Closure
- I set of items of G
- Closure(I)
  - Initially I
  - Repeat
    - If A->α.B|γ in closure(I) and B->γ is a production add B->γ to closure
  - Until no new items get added
Goto

- Goto(I,X), I set of items, X grammar symbol
- Goto(I,X) := closure({A->αX.|β| A->α.X|β| ∈ I})
- For valid items I for viable prefix γ, then goto(I,X) = valid items for viable prefix γX

Sets of Items Construction

procedure items(G');
begin
  C := closure({S->.S});
  repeat
    for each set of items I in C and each grammar symbol X such that goto(I,X) is not empty and not in C do
      add goto(I,X) to C
    until no more changes to C
end
SLR Table Construction

Input \( G' \) augmented grammar

1. Construct \( C = \{ I_0, \ldots, I_n \} \), collection of LR(0) items
2. State \( I_i \) is constructed from \( I = I_i \) with parsing actions:
   a) if \( A \rightarrow \alpha.a.\beta \) is in \( I \) and \( \text{goto}(I_i, a) = I_j \), then \( \text{action}[i, a] \) := shift \( j \) (a must be a terminal)
   b) if \( A \rightarrow \alpha \) is in \( I \) and \( \text{follow}(A) \) then \( \text{action}[i, a] \) := reduce \( A \rightarrow \alpha \) for all \( a \) in \( \text{follow}(A) \)
   c) if \( S' \rightarrow \alpha \) is in \( I \) then set \( \text{action}[I_i, S] = \text{accept} \)
3. \( \text{Goto} \) transitions for state \( I \) are constructed for all nonterminal \( A \) as follows: if \( \text{goto}(I_i, A) = I_j \), then \( \text{goto}[I_i, A] := j \)
4. Undefined entries are set to error
5. The initial state of the parser is the on constructed from the items containing \( S' \rightarrow .S \)

SLR Table Example

Same grammar...

Parsing Example

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ACTION