Parsing - Roadmap

- Parser:
  - decision procedure: builds a parse tree
- Top-down vs. bottom-up
- LL(1) – Deterministic Parsing
  - recursive-descent
  - table-driven
- LR(k) – Deterministic Parsing
  - LR(0), SLR(1), LR(1), LALR(1)
- Parsing arbitrary CFGs – Polynomial time parsing
Top-Down vs. Bottom Up

Grammar: \[ S \rightarrow A \ B \]
\[ A \rightarrow c \mid \epsilon \]
\[ B \rightarrow cb \mid ca \]

Input String: ccbca

<table>
<thead>
<tr>
<th>Top-Down/leftmost</th>
<th>Bottom-Up/rightmost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ⇒ AB</td>
<td>S⇒AB</td>
</tr>
<tr>
<td>⇒ cB</td>
<td>A⇒c</td>
</tr>
<tr>
<td>⇒ ccbB</td>
<td>B⇒cbB</td>
</tr>
<tr>
<td>⇒ ccbca</td>
<td>B⇒ca</td>
</tr>
</tbody>
</table>

Bottom-up parsing overview

• Start from terminal symbols, search for a path to the start symbol
• Apply shift and reduce actions: postpone decisions
• LR parsing:
  – L: left to right parsing
  – R: rightmost derivation (in reverse or bottom-up)
• LR(0) → SLR(1) → LR(1) → LALR(1)
  – 0 or 1 or \( k \) lookahead symbols
Actions in Shift-Reduce Parsing

- **Shift**
  - add terminal to parse stack, advance input

- **Reduce**
  - If $\alpha w$ on stack, and $A \rightarrow w$, and there is a $\beta \in T^*$ such that $S \Rightarrow^*_{rm} \alpha A\beta \Rightarrow_{rm} \alpha w\beta$ then we can prune the handle $w$; we reduce $\alpha w$ to $\alpha A$ on the stack
  - $\alpha w$ is a viable prefix

- **Error**
- **Accept**

Questions

- **When to shift/reduce?**
  - What are valid handles?
  - Ambiguity: Shift/reduce conflict

- **If reducing, using which production?**
  - Ambiguity: Reduce/reduce conflict
Rightmost derivation for
\[ \text{id + id * id} \]

\[
\begin{align*}
E & \rightarrow E + E & E & \Rightarrow E * E \\
E & \rightarrow E * E & \Rightarrow E * \text{id} \\
E & \rightarrow (E) & \Rightarrow E + E * \text{id} \\
E & \rightarrow -E & \Rightarrow E + \text{id} * \text{id} & \text{reduce with } E \rightarrow \text{id} \\
E & \rightarrow \text{id} & \Rightarrow \text{id} + \text{id} * \text{id} & \text{shift}
\end{align*}
\]

\[
E \Rightarrow_{\text{rm}}^* E + E \backslash \text{id}
\]

LR Parsing

- Table-based parser
  - Creates rightmost derivation (in reverse)
  - For “less massaged” grammars than LL(1)
- Data structures:
  - Stack of states/symbols \{s\}
  - Action table: \text{action}[s, a]; a \in T
  - Goto table: \text{goto}[s, X]; X \in N
### Productions

1. \( T \rightarrow F \)
2. \( T \rightarrow TF \)
3. \( F \rightarrow id \)
4. \( F \rightarrow (T) \)

### Action/Goto Table

<table>
<thead>
<tr>
<th>*</th>
<th>( )</th>
<th>id</th>
<th>$</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>1</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td>Acc!</td>
</tr>
<tr>
<td>2</td>
<td>S5</td>
<td>S8</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S5</td>
<td>S8</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>5</td>
<td>S5</td>
<td>S8</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S3</td>
<td>S7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>8</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
</tbody>
</table>

### Trace “(id)*id”

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(id)*id$</td>
<td>Shift S5</td>
</tr>
<tr>
<td>0 5</td>
<td>id)*id$</td>
<td>Shift S8</td>
</tr>
<tr>
<td>0 5 8</td>
<td>)*id$</td>
<td>Reduce 3 F→id, pop 8, goto [5,F]=1</td>
</tr>
<tr>
<td>0 5 1</td>
<td>)*id$</td>
<td>Reduce 1 T→F, pop 1, goto [5,T]=6</td>
</tr>
<tr>
<td>0 5 6</td>
<td>)*id$</td>
<td>Shift S7</td>
</tr>
<tr>
<td>0 5 6 7</td>
<td>*id$</td>
<td>Reduce 4 F→(T), pop 7 6 5, goto [0,F]=1</td>
</tr>
<tr>
<td>0 1</td>
<td>*id$</td>
<td>Reduce 1 T→F, pop 1, goto [0,T]=2</td>
</tr>
</tbody>
</table>
Trace "(id)*id"

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>* id $</td>
<td>Reduce 1 T→F, pop 1, goto [0,T]=2</td>
</tr>
<tr>
<td>0 2</td>
<td>* id $</td>
<td>Shift S3</td>
</tr>
<tr>
<td>0 2 3</td>
<td>* id $</td>
<td>Shift S8</td>
</tr>
<tr>
<td>0 2 3 8</td>
<td>$</td>
<td>Reduce 3 F→id, pop 8, goto [3,F]=4</td>
</tr>
<tr>
<td>0 2 3 4</td>
<td>$</td>
<td>Reduce 2 T→T*F, pop 4 3 2, goto [0,T]=2</td>
</tr>
<tr>
<td>0 2</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Trace "(id)*id"

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(id)*id</td>
<td></td>
</tr>
</tbody>
</table>

Productions

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T→F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T→T*F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F→id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F→(T)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>S5</td>
<td>S8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S5</td>
<td>S8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>4</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>S5</td>
<td>S8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S5</td>
<td>S8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>4</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
</tbody>
</table>

10/18/07
### Tracing LR: action[s, a]

- **case shift** u:
  - push state u
  - read new a

- **case reduce** r:
  - lookup production r: X → Y₁..Yₖ;
  - pop k states, find state u
  - push goto[u, X]

- **case accept**: done
- **no entry in action table**: error
Configuration set

- Each set is a parser state
- We use the notion of a dotted rule or item:
  \[ T \rightarrow T \ast \cdot F \]
- The dot is before \( F \), so we predict all rules with \( F \) as the left-hand side
  \[ T \rightarrow T \ast \cdot F \]
  \[ F \rightarrow \cdot ( T ) \]
  \[ F \rightarrow \cdot \text{id} \]
- This creates a configuration set (or item set)

Closure

Closure property:
- If \( T \rightarrow X_1 \ldots X_i \cdot X_{i+1} \ldots X_n \) is in set, and \( X_{i+1} \) is a nonterminal, then \( X_{i+1} \rightarrow \cdot Y_1 \ldots Y_m \) is in the set as well for all productions \( X_{i+1} \rightarrow Y_1 \ldots Y_m \)
- Compute as fixed point
- The closure property creates a configuration set (item set) from a dotted rule (item).
Starting Configuration

• Augment Grammar with S’
• Add production S’ → S
• Initial configuration set is
  closure(S’ → • S)

Example: I = closure(S’ → • T)

S’ → • T
T → • T * F
T → • F
F → • id
F → • ( T )

S’ → T
T → F | T * F
F → id | ( T )
Successor(I, X)

Informally: “move by symbol X”
1. move dot to the right in all items where
dot is before X
2. remove all other items
   (viable prefixes only!)
3. compute closure

Successor Example

\[
\begin{align*}
I &= \{ S' \rightarrow \bullet T, \\
    &\quad T \rightarrow \bullet F, \\
    &\quad T \rightarrow \bullet T \ast F, \\
    &\quad F \rightarrow \bullet \text{id}, \\
    &\quad F \rightarrow \bullet ( T ) \} \\
\text{Compute Successor}(I, "\(")
\end{align*}
\]

\[
\{ F \rightarrow ( \bullet T ), T \rightarrow \bullet F, T \rightarrow \bullet T \ast F, \\
    F \rightarrow \bullet \text{id}, F \rightarrow \bullet ( T ) \}
\]
Sets-of-Items Construction

Family of configuration sets

function items(G')
    C = { closure({S' → • S}) };
    do foreach I ∈ C do
        foreach X ∈ (N U T) do
            C = C ∪ { Successor(I, X) };
    while C changes;

0: S' → • T
   T → • F
   T → • T * F
   F → • id
   F → • ( T )

1: T → F •
2: S' → T •
   T → T * F •
3: T → • F
   F → • id
   F → • ( T )
4: T → T * F •
5: F → ( • T )
   T → • F
   T → • T * F
   F → • id
   F → • ( T )
6: F → ( T • )
7: F → ( T ) •
8: F → id •
LR(0) Construction

1. Construct \( F = \{ I_0, I_1, \ldots I_n \} \)
2. a) if \( \{ A \rightarrow \alpha \} \in I_i \) and \( A \neq S' \)
   then \( \text{action}[i, \_] := \text{reduce } A \rightarrow \alpha \)
   b) if \( \{ S' \rightarrow S \} \in I_i \)
   then \( \text{action}[i,\$] := \text{accept} \)
   c) if \( \{ A \rightarrow \alpha \cdot a \beta \} \in I_i \) and \( \text{Successor}(I_i,a) = I_j \)
   then \( \text{action}[i,a] := \text{shift } j \)
3. if \( \text{Successor}(I_i,A) = I_j \) then \( \text{goto}[i,A] := j \)
LR(0) Construction (cont’d)

4. All entries not defined are errors
5. Make sure \( I_0 \) is the initial state

- Note: LR(0) always reduces if \( \{ A \to \alpha \circ \} \in I_i \), no lookahead
- Shift and reduce items can’t be in the same configuration set
  - Accepting state doesn’t count as reduce item
- At most one reduce item per set

Set-of-items with Epsilon rules
LR(0) conflicts:

\[
egin{align*}
S' & \rightarrow T \\
T & \rightarrow F \\
T & \rightarrow T \ast F \\
T & \rightarrow \text{id} \\
F & \rightarrow \text{id} \mid (T) \\
F & \rightarrow \text{id} = T \\
\end{align*}
\]

11: \( F \rightarrow \text{id} \cdot \)  \\
\( F \rightarrow \text{id} \cdot = T \)  \\
Shift/reduce conflict

1: \( F \rightarrow \text{id} \cdot \)  \\
\( T \rightarrow \text{id} \cdot \)  \\
Reduce/Reduce conflict

Need more lookahead: SLR(1)

LR(0) Grammars

- An LR(0) grammar is a CFG such that the LR(0) construction produces a table without conflicts (a deterministic pushdown automata)
- \( S \Rightarrow^*_{\text{rm}} \alpha A \beta \Rightarrow_{\text{rm}} \alpha w \beta \) and \( A \rightarrow w \) then we can prune the handle \( w \)
  - pruning the handle means we can reduce \( \alpha w \) to \( \alpha A \) on the stack
- Every viable prefix \( \alpha w \) can recognized using the DFA built by the LR(0) construction
LR(0) Grammars

- Once we have a viable prefix on the stack, we can prune the handle and then restart the DFA to obtain another viable prefix, and so on ...
- In LR(0) pruning the handle can be done without any look-ahead
  - this means that in the rightmost derivation,
  - \( S \Rightarrow r_m \alpha A \beta \Rightarrow r_m \alpha w \beta \) we reduce using a unique rule \( A \rightarrow w \) without ambiguity, and without looking at \( \beta \)
- No ambiguous context-free grammar can be LR(0)

LR(0) Grammars \( \subset \) Context-free Grammars

---

SLR(1) : Simple LR(1) Parsing

\[0: S' \rightarrow \bullet T \]
\[T \rightarrow \bullet F \]
\[T \rightarrow \bullet T * F \]
\[T \rightarrow \bullet C (T) \]
\[F \rightarrow \bullet id \]
\[F \rightarrow \bullet id ++ \]
\[F \rightarrow \bullet ( T ) \]
\[C \rightarrow \bullet id \]

\[1: F \rightarrow \bullet id \]
\[F \rightarrow \bullet id ++ \]
\[C \rightarrow \bullet id \]

Follow(\( F \)) = \{ *, ), $ \}
Follow(\( C \)) = \{ ( \} 

action[1,*] = action[1,.)] = action[1,$] = Reduce \( F \rightarrow id \)
action[1,(] = Reduce \( C \rightarrow id \)
action[1,++] = Shift
SLR(1) Construction

1. Construct $F = \{I_0, I_1, \ldots, I_n\}$
2. a) if $\{A \to \alpha \star \} \in I_i$ and $A \neq S'$
   
   then $\text{action}[i, b] := \text{reduce } A \to \alpha$
   
   for all $b \in \text{Follow}(A)$

   b) if $\{S' \to S \star \} \in I_i$

   then $\text{action}[i, S] := \text{accept}$

   c) if $\{A \to \alpha \star a \beta\} \in I_i$ and $\text{Successor}(I_i, a) = I_j$

   then $\text{action}[i, a] := \text{shift } j$

3. if $\text{Successor}(I_i, A) = I_j$ then $\text{goto}[i, A] := j$

SLR(1) Construction (cont’d)

4. All entries not defined are errors
5. Make sure $I_0$ is the initial state

- Note: SLR(1) only reduces
  $\{A \to \alpha \star \}$ if lookahead in $\text{Follow}(A)$
- Shift and reduce items or more than one reduce item can be in the same configuration set as long as lookaheads are disjoint
SLR(1) Conditions

• A grammar is SLR(1) if for each configuration set:
  – For any item \{A \rightarrow \alpha \cdot x \beta: x \in T\} there is no \{B \rightarrow \gamma \cdot: x \in \text{Follow}(B)\}
  – For any two items \{A \rightarrow \alpha \cdot\} and \{B \rightarrow \beta \cdot\}
    \text{Follow}(A) \cap \text{Follow}(B) = \emptyset

LR(0) Grammars \subset SLR(1) Grammars

Is this grammar SLR(1)?
SLR limitation: lack of context

Input: \text{id} = \text{id}

Follow(R) = \{ =, $ \}

Solution: Canonical LR(1)

- Extend definition of configuration
  - Remember lookahead
- New closure method
- Extend definition of Successor
LR(1) Configurations

• \([A \rightarrow \alpha \cdot \beta, a] \) for \(a \in T\) is valid for a viable prefix \(\delta \alpha\) if there is a rightmost derivation:
  \(S \Rightarrow^* \delta A \eta \Rightarrow^* \delta \alpha \beta \eta\) and
  \((\eta = a \gamma)\) or \((\eta = \varepsilon\) and \(a = \$$)\)

• Notation: \([A \rightarrow \alpha \cdot \beta, a/b/c]\)
  – if \([A \rightarrow \alpha \cdot \beta, a]\), \([A \rightarrow \alpha \cdot \beta, b]\), \([A \rightarrow \alpha \cdot \beta, c]\)
    are valid configurations

LR(1) Configurations

\[
\begin{align*}
S & \rightarrow B \hspace{1em} B \\
B & \rightarrow a \hspace{1em} B \mid b \\
\text{• } S & \Rightarrow^*_{\text{rm}} aaBab \Rightarrow_{\text{rm}} aaaBab \\
\text{• Item } [B \rightarrow a \cdot B, a] \text{ is valid for viable prefix } aaa \\
\text{• } S & \Rightarrow^*_{\text{rm}} BaB \Rightarrow_{\text{rm}} BaaB \\
\text{• Also, item } [B \rightarrow a \cdot B, \$$] \text{ is valid for viable prefix } Baa
\end{align*}
\]
LR(1) Closure

Closure property:
• If \([A \rightarrow \alpha \cdot B\beta, a]\) is in set, then
  \([B \rightarrow \cdot \gamma, b]\) is in set if \(b \in \text{First}(\beta a)\)
• Compute as fixed point
• Only include contextually valid lookaheads to guide reducing to B

Starting Configuration

• Augment Grammar with \(S'\) just like for LR(0), SLR(1)
• Initial configuration set is
  \(I = \text{closure}([S' \rightarrow \cdot S, \$])\)
Example: closure([S’ → • S, $])

[ S’ → • S, $]
[S → • L = R, $]
[S → • R, $]
[L → • * R, =]
[L → • id, =]
[R → • L, $]
[L → • *R, $]
[L → • id, $]

LR(1) Successor(C, X)

• Let I = [A → α•Bβ, a] or [A → α•bβ, a]
• Successor(I, B)
  = closure([A → αB • β, a])
• Successor(I, b)
  = closure([A → αb • β, a])
LR(1) Example

0: \( S' \rightarrow S, \) $
  S \rightarrow L = R, $
  S \rightarrow R, $
  L \rightarrow * R, =/\$
  L \rightarrow \text{id}, =/\$
  R \rightarrow L, $

1: L \rightarrow \text{id} \cdot, $/=

2: S \rightarrow L \cdot = R, $
  R \rightarrow L \cdot, $

3: S \rightarrow L = R, $
  R \rightarrow L, $
  L \rightarrow * R, $
  L \rightarrow \text{id}, $

4: L \rightarrow \text{id} \cdot, $

5: R \rightarrow L \cdot, $

6: S \rightarrow L = R \cdot, $

7: S' \rightarrow S \cdot, $

8: L \rightarrow * \cdot R, $
  R \rightarrow L, $
  L \rightarrow * R, $
  L \rightarrow \text{id}, $

9: L \rightarrow * R \cdot, $

LR(1) Example (contd)

3: S \rightarrow L = \cdot R, $
  R \rightarrow \cdot L, $
  L \rightarrow * R, $
  L \rightarrow \text{id}, $

4: L \rightarrow \text{id} \cdot, $

5: R \rightarrow L \cdot, $

8: L \rightarrow * \cdot R, $
  R \rightarrow \cdot L, $
  L \rightarrow * R, $
  L \rightarrow \text{id}, $

9: L \rightarrow * R \cdot, "$
LR(1) Example (contd)

0: $S' \rightarrow \bullet S, \$ 
   S \rightarrow \bullet L = R, \$ 
   S \rightarrow \bullet R, \$ 
   L \rightarrow \bullet * R, =/$ 
   L \rightarrow \bullet id, =/$ 
   R \rightarrow \bullet L, \$

1: L \rightarrow id, =/

2: S \rightarrow R, \$

3: L \rightarrow \bullet L, =/
   L \rightarrow \bullet * R, =/
   L \rightarrow \bullet id, =/

4: R \rightarrow \bullet L, =/
   L \rightarrow \bullet * R, =/

5: S \rightarrow R\bullet, \$

6: R \rightarrow L, =/

7: Acc

8: S \rightarrow L = R 
   R \rightarrow L, =/

9: S \rightarrow R

10: S \rightarrow L = R 
    R \rightarrow id, =/

11: R \rightarrow L, =/

12: L \rightarrow \bullet * R, =/

13: R \rightarrow L, =/

Productions

- 1: $S \rightarrow L = R$
- 2: $S \rightarrow R$
- 3: $L \rightarrow \bullet * R$
- 4: $L \rightarrow \bullet id$
- 5: $R \rightarrow L$

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>=</th>
<th>*</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td></td>
<td></td>
<td>S10</td>
<td>7</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>R4</td>
<td></td>
<td></td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S3</td>
<td></td>
<td></td>
<td>R5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S4</td>
<td></td>
<td></td>
<td>S8</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S1</td>
<td></td>
<td></td>
<td>S10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td></td>
<td></td>
<td>R5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>R3</td>
<td></td>
<td></td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LR(1) Construction

1. Construct \( F = \{ I_0, I_1, \ldots I_n \} \)
2. a) if \([A \rightarrow \alpha\bullet, a] \in I_i\) and \(A \neq S'\)
    then \(\text{action}[i, a] := \text{reduce}\ A \rightarrow \alpha\)
   b) if \([S' \rightarrow S\bullet, \$] \in I_i\)
   then \(\text{action}[i, \$] := \text{accept}\)
   c) if \([A \rightarrow \alpha\bullet a\beta, b] \in I_i\) and \(\text{Successor}(I_i, a) = I_j\)
   then \(\text{action}[i, a] := \text{shift} j\)
3. if \(\text{Successor}(I_i, A) = I_j\) then \(\text{goto}[i, A] := j\)

LR(1) Construction (cont’d)

4. All entries not defined are errors
5. Make sure \(I_0\) is the initial state

- Note: LR(1) only reduces using \(A \rightarrow \alpha\) for \([A \rightarrow \alpha\bullet, a]\) if a follows
- LR(1) states remember context by virtue of lookahead
- Possibly many states!
  - LALR(1) combines some states
LR(1) Conditions

• A grammar is LR(1) if for each configuration set holds:
  – For any item \([A \rightarrow \alpha x \beta, a]\) with \(x \in T\) there is no
    \([B \rightarrow \gamma \cdot, x]\)
  – For any two complete items \([A \rightarrow \gamma \cdot, a]\) and
    \([B \rightarrow \beta \cdot, b]\) it follows \(a\) and \(a \neq b\).

• Grammars:
  – \(LR(0) \subset SLR(1) \subset LR(1) \subset LR(k)\)

• Languages expressible by grammars:
  – \(LR(0) \subset SLR(1) \subset LR(1) = LR(k)\)

Canonical LR(1) Recap

• LR(1) uses left context, current handle and lookahead to decide when to reduce or shift
• Most powerful parser so far
• LALR(1) is practical simplification with fewer states
Merging States in LALR(1)

• \( S' \rightarrow S \)
  \( S \rightarrow XX \)
  \( X \rightarrow aX \)
  \( X \rightarrow b \)

• **Same Core Set**

• **Different lookaheads**

<table>
<thead>
<tr>
<th>3: ( X \rightarrow a \cdot X, a/b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \rightarrow \cdot a X, a/b )</td>
</tr>
<tr>
<td>( X \rightarrow \cdot b, a/b )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6: ( X \rightarrow a \cdot X, ) $</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \rightarrow \cdot a X, ) $</td>
</tr>
<tr>
<td>( X \rightarrow \cdot b, ) $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>36: ( X \rightarrow a \cdot X, a/b/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \rightarrow \cdot a X, a/b/$</td>
</tr>
<tr>
<td>( X \rightarrow \cdot b, a/b/$</td>
</tr>
</tbody>
</table>

R/R conflicts when merging

• \( B \rightarrow d \)
  \( B \rightarrow f X g \)
  \( X \rightarrow ... \)

• If R/R conflicts are introduced, grammar is not LALR(1)!

<table>
<thead>
<tr>
<th>2: ( B \rightarrow d \cdot, c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B \rightarrow f X g \cdot, e )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4: ( B \rightarrow d \cdot, g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B \rightarrow f X g \cdot, c )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24: ( B \rightarrow d \cdot, c/g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B \rightarrow f X g \cdot, c/e )</td>
</tr>
</tbody>
</table>
LALR(1)

- LALR(1) Condition:
  - Merging in this way does not introduce reduce/reduce conflicts
  - Shift/reduce can’t be introduced
- Merging brute force or step-by-step
- More compact than canonical LR, like SLR(1)
- More powerful than SLR(1)
  - Not always merge to full Follow Set

S/R & ambiguous grammars

- Lx(k) Grammar vs. Language
  - Grammar is Lx(k) if it can be parsed by Lx(k) method – according to criteria that is specific to the method.
  - A Lx(k) grammar may or may not exist for a language.
- Even if a given grammar is not LR(k), shift/reduce parser can sometimes handle them by accounting for ambiguities
  - Example: ‘dangling’ else
    - Preferring shift to reduce means matching inner ‘if’
Dangling ‘else’

1. \( S \rightarrow \text{if } E \text{ then } S \)
2. \( S \rightarrow \text{if } E \text{ then } S \text{ else } S \)

- Viable prefix “if E then if E then S”
  - Then read else
- Shift “else” (means go for 2)
- Reduce (reduce using production #1)
- NB: dangling else as written above is ambiguous
  - NB: Ambiguity can be resolved, but there’s still no LR(k) grammar

Precedence & Associativity

- Consider \( E \rightarrow E - E | E * E | \text{id} \)

<table>
<thead>
<tr>
<th>Reduce</th>
<th>Shift</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>( E )</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>( E )</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>( E - E \cdot \ast )</td>
<td>( E )</td>
<td>( E - E \cdot \ast )</td>
</tr>
<tr>
<td>( \text{id - id} \ast \text{id} )</td>
<td>( \text{id - id} \ast \text{id} )</td>
<td>( \text{id - id} \ast \text{id} )</td>
</tr>
</tbody>
</table>
Precedence Relations

• Let $A \rightarrow w$ be a rule in the grammar
• And $b$ is a terminal
• In some state $q$ of the LR(1) parser there is a shift-reduce conflict:
  – either reduce with $A \rightarrow w$ or shift on $b$
• Write down a rule, either:
  $A \rightarrow w, < b$ or $A \rightarrow w, > b$

Precedence Relations

• $A \rightarrow w, < b$ means rule has less precedence and so we shift if we see $b$ in the lookahead
• $A \rightarrow w, > b$ means rule has higher precedence and so we reduce if we see $b$ in the lookahead
• If there are multiple terminals with shift-reduce conflicts, then we list them all:
  $A \rightarrow w, > b, < c, > d$
Precedence Relations

- Consider the grammar
  \[ E \rightarrow E + E | E * E | ( E ) | a \]
- Assume left-association so that \( E+E+E \) is interpreted as \((E+E)+E\)
- Assume multiplication has higher precedence than addition
- Then we can write precedence rules/relns:
  \[ E \rightarrow E + E, > +, < * \]
  \[ E \rightarrow E * E, > +, > * \]

Precedence & Associativity

\[ E \rightarrow E + E, > +, < * \]
\[ E \rightarrow E * E, > +, > * \]
Handling S/R & R/R Conflicts

• Have a conflict?
  – No? – Done, grammar is compliant.
• Already using most powerful parser available?
  – No? – Upgrade and goto 1
• Can the grammar be rearranged so that the conflict disappears?
  – While preserving the language!

Conflicts revisited (cont’d)

• Can the grammar be rearranged so that the conflict disappears?
  – No?
    • Is the conflict S/R and does shift-to-reduce preference yield desired result?
      – Yes: Done. (Example: dangling else)
    • Else: Bad luck
  – Yes: Is it worth it?
    • Yes, resolve conflict.
    • No: live with default or specified conflict resolution (precedence, associativity)
Compiler (parser) compilers

- Rather than build a parser for a particular grammar (e.g. recursive descent), write down a grammar as a text file
- Run through a compiler compiler which produces a parser for that grammar
- The parser is a program that can be compiled and accepts input strings and produces user-defined output

- For LR parsing, all it needs to do is produce action/goto table
  - Yacc (yet another compiler compiler) was distributed with Unix, the most popular tool. Uses LALR(1).
  - Many variants of yacc exist for many languages
- As we will see later, translation of the parse tree into machine code (or anything else) can also be written down with the grammar
- Handling errors and interaction with the lexical analyzer have to be precisely defined
Parsing - Summary

- Top-down vs. bottom-up
- Lookahead: FIRST and FOLLOW sets
- LL(1) – Parsing: $O(n)$ time complexity
  - recursive-descent and table-driven predictive parsing
- LR(k) – Parsing: $O(n)$ time complexity
  - LR(0), SLR(1), LR(1), LALR(1)
- Resolving shift/reduce conflicts
  - using precedence, associativity