Bottom Up Parsing

PITT CS 2210
Bottom Up Parsing

- More powerful than top down
  - Don’t need left factored grammars
  - Can handle left recursion
  - Can express a larger set of languages

- Begins at leaves and works to the top
  - In reverse order of rightmost derivation
    (In effect, builds tree from left to right)

- Also known as Shift-Reduce parsing
  - Involves two types of operations: shift and reduce
Parser Implementation

Parser Stack – holds consumed portion of derivation string
Table – “actions” to perform based on (stack top, current token)
Parser Driver – table-driven code to perform actions in table given
stack top and current token
Parser Implementation

Actions
1. Shift – consume input symbol and add symbol onto the stack
2. Reduce – replace an RHS at top of stack to LHS of a production rule, reducing stack contents
3. Accept – success (when reduced to start symbol and input at $)
4. Error
Z → b M b
M → ( L | a
L → M a ) | )

Considering string: w = b ( a a ) b

The rightmost derivation of this parse tree:
Z ⇒ b M b ⇒ b ( L b ⇒ b ( M a ) b ⇒ b ( a a ) b

Bottom up parsing involves finding “handles” (RHSs) to reduce
b ( a a ) b ⇒ b ( M a ) b ⇒ b ( L b ⇒ b M b ⇒ Z
Handle

- Informally: RHS of a production rule that, when replaced with LHS, will lead to the start symbol

- Definition:
  - **Sentential form**: Any string derivable from the start symbol, comprised of terminals and non-terminals
  - Let $\alpha\beta w$ be a sentential form where
    - $\alpha$ is an arbitrary string of symbols
    - $X \rightarrow \beta$ is a production
    - $w$ is a string of terminals (already derived portion)
    Then $\beta$ is a *handle* of $\alpha\beta w$ if
      - $S \Rightarrow^* \alpha X w \Rightarrow \alpha \beta w$ by a rightmost derivation
  - Handles formalize the intuition “$\beta$ should be reduced to $X$ for a successful parse”, but does not really say how to find them
Handle Always Occurs at Top of Stack

Z → b M b
M → ( L | a
L → M a ) | )

String
b ( a a ) $
## Handle Always Occurs at Top of Stack

### Grammar

- $E \rightarrow E + E$
- $E \rightarrow E \ast E$
- $E \rightarrow (E)$
- $E \rightarrow id$

### Handle Production

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Handle</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id_1 + id_2 \ast id_3$</td>
<td>$id_1$</td>
<td>$E \rightarrow id$</td>
</tr>
<tr>
<td>$E + id_2 \ast id_3$</td>
<td>$id_2$</td>
<td>$E \rightarrow id$</td>
</tr>
<tr>
<td>$E + E \ast id_3$</td>
<td>$id_3$</td>
<td>$E \rightarrow id$</td>
</tr>
<tr>
<td>$E + E \ast E$</td>
<td>$E \ast E$</td>
<td>$E \rightarrow E \ast E$</td>
</tr>
<tr>
<td>$E + E$</td>
<td>$E + E$</td>
<td>$E \rightarrow E + E$</td>
</tr>
</tbody>
</table>

### #: location of handle

- Left of #: completely reduced (except for handle), Right of #: unreduced
- $id_1 # + id_2 \ast id_3 \Rightarrow E # + id_2 \ast id_3 \Rightarrow E + # id_2 \ast id_3 \Rightarrow E + id_2 # \ast id_3$
- $\Rightarrow E + E # \ast id_3 \Rightarrow E + E \ast id_3 # \Rightarrow E + E \ast E # \Rightarrow E + E # \Rightarrow E$

### # moves monotonically from left to right (reverse of rightmost derivation)

- If we shift judiciously to always maintain string left of # on the stack, handle always occurs at top of stack (never in the middle)
Handle Always Occurs at Top of Stack

Consider our usual grammar

\[
E \rightarrow T + E | T \\
T \rightarrow \text{int} * T | \text{int} | ( E )
\]

Consider the string: int * int + int

<table>
<thead>
<tr>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int # + int</td>
<td>T \rightarrow \text{int}</td>
</tr>
<tr>
<td>int * T # + int</td>
<td>T \rightarrow \text{int} * T</td>
</tr>
<tr>
<td>T + int #</td>
<td>T \rightarrow \text{int}</td>
</tr>
<tr>
<td>T + T #</td>
<td>E \rightarrow T</td>
</tr>
<tr>
<td>T + E #</td>
<td>E \rightarrow T + E</td>
</tr>
<tr>
<td>E #</td>
<td></td>
</tr>
</tbody>
</table>

- Essentially, a depth-first left-to-right traversal of parse tree
- Makes life easier for parser (no need to access middle of stack)
Ambiguous Grammars

Conflicts arise with ambiguous grammars

Just like LL parsing, bottom up parsing tries to predict the correct action, but if there are multiple correct actions, conflicts arise

Example:

Consider the ambiguous grammar

\[ E \rightarrow E \ast E \mid E + E \mid (E) \mid \text{int} \]

<table>
<thead>
<tr>
<th>Sentential form</th>
<th>Actions</th>
<th>Sentential form</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int \ast int + int</td>
<td>shift</td>
<td>int \ast int + int</td>
<td>shift</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>E \ast E # + int</td>
<td>reduce E \rightarrow E \ast E</td>
<td>E \ast E # + int</td>
<td>shift</td>
</tr>
<tr>
<td>E # + int</td>
<td>shift</td>
<td>E # + # int</td>
<td>shift</td>
</tr>
<tr>
<td>E + # int</td>
<td>shift</td>
<td>E + int #</td>
<td>reduce E \rightarrow int</td>
</tr>
<tr>
<td>E + E #</td>
<td>reduce E \rightarrow E + E</td>
<td>E * E + # int</td>
<td>shift</td>
</tr>
<tr>
<td>E #</td>
<td>reduce E \rightarrow E + E</td>
<td>E * E + int #</td>
<td>shift</td>
</tr>
<tr>
<td></td>
<td>reduce E \rightarrow E * E</td>
<td>E * E #</td>
<td>reduce E \rightarrow int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E #</td>
<td>reduce E \rightarrow E + E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reduce E \rightarrow E * E</td>
</tr>
</tbody>
</table>
Ambiguity

- In the first step shown, we can either shift or reduce by
  
  \[ E \rightarrow E \times E \]

  Choice because of precedence of + and *

  Same problem with association of * and +

- Can always rewrite to encode precedence and associativity
  
  - Can sometimes result in convoluted grammars
  - Tools have other means to encode precedence and association

- But must get rid of remaining ambiguity (e.g. if-then-else)
  
  - Ambiguity show up as “conflicts” in the parsing table
    (Just like in tables for LL parsers)
Properties of Bottom Up Parsing

- Handles always appear at the top of the stack
  - Never in middle of stack
  - Justifies use of stack in shift – reduce parsing (just like for Top-Down!)

- Results in an easily generalized shift – reduce strategy
  - If there is no handle at the top of the stack, shift
  - If there is a handle, reduce to the non-terminal
  - Easy to automate the synthesis of the parser using a table

- Can have conflicts
  - If it is legal to either shift or reduce then there is a shift-reduce conflict.
  - If there are two legal reductions, then there is a reduce-reduce conflict.
  - Most often occur because of ambiguous grammars
    - In rare cases, because of non-ambiguous grammars not amenable to parser
Types of Bottom Up Parsers

- Types of bottom up parsers
  - Simple precedence parsers
  - Operator precedence parsers
  - LR family parsers
  - …

- In this course, we will only discuss LR family parsers
  - Most automated tools generate either LL or LR parsers
LR Parsers

- LR family of parsers
  - LR(k)  L – left to right
    R – rightmost derivation in reverse
    k elements of look ahead

- Attractive
  1. More powerful than LL(k)
     - Handles more grammars: no left recursion removal, left factoring needed
     - Handles more (and most practical) languages: \( LL(k) \subset LR(k) \)
  2. Efficient as LL(k)
     - Linear in time and space to length of input (same as LL(k))
  3. Convenient as LL(k)
     - Can generate automatically from grammar – YACC, Bison

- Less attractive – more complex than LL(k) parser
Implementation

-- LR Parsing

Pitt, CS 2210
Viable Prefix

- Definition: $\alpha$ is a viable prefix if
  - $\alpha$ is a prefix of a rightmost sentential form (there exists a $w$ where $\alpha w$ is a rightmost sentential form)
  - $\alpha$ only includes symbols up to the “handle”
  - In other words, if there exists a $w$ where $\alpha#w$ is a configuration of a shift-reduce parser
    
    $b(a#a)b \Rightarrow b(M#a)b \Rightarrow b(L#b \Rightarrow bM#b \Rightarrow Z)$

- If stack contains a viable prefix, parser is on the right track
  - Has a chance of accepting depending on remaining input

- Shift-reduce parsing is the process of massaging the contents of the parse stack from viable prefix to viable prefix
  - Reject if neither shifting or reducing results in a viable prefix
Massaging into a Viable Prefix

- How do you know what results in a viable prefix?
  - Example grammar
    
    \[
    S \rightarrow a \ B \ S \mid b \\
    B \rightarrow b
    \]

  - Example shift and reduce on: \(a \# \ b \ b\)
    
    
    Shift: \(a \# \ b \ b \Rightarrow a \ b \ # \ b\)  
    Does shifting result in a viable prefix?
    
    Reduce: \(a \ b \ # \ b \Rightarrow a \ B \ # \ b\)  
    Should I apply \(B \rightarrow b\) (and not \(S \rightarrow b\))?  

- Keep track of where you are in the current production
Massaging into a Viable Prefix

- Example grammar
  
  \[ S \rightarrow a\ B\ S \mid b \]
  
  \[ B \rightarrow b \]

- Let a dot (\(\cdot\)) indicate extent of production already seen

  - **Shift ‘b’:** \(a \# b\ b \Rightarrow a\ b\ \#\ b\) (Why?)
    
    Because we need a ‘B’ and we are just about to start with ‘B’
    
    \[ S \rightarrow a\ .\ B\ S,\ B \rightarrow .\ b \Rightarrow \]
    
    \[ S \rightarrow a\ .\ B\ S,\ B \rightarrow .\ b .\]  

  - **Reduce ‘\(B \rightarrow b\)’:** \(a\ b\ \#\ b \Rightarrow a\ B\ \#\ b\) (Why?)
    
    Because we need a ‘B’ and we’ve just finished completing ‘B’
    
    \[ S \rightarrow a\ .\ B\ S,\ B \rightarrow b\ .\ \Rightarrow \]
    
    \[ S \rightarrow a\ B\ .\ S,\ S \rightarrow .\ a\ B\ S,\ S \rightarrow .\ b \]
LR(0) Item Notation

- LR(0) Item: a production + a dot on the RHS
  - Dot indicates extent of production already seen
  - In example grammar
    - Items for production $S \rightarrow a B S$
      - $S \rightarrow . a B S$
      - $S \rightarrow a . B S$
      - $S \rightarrow a B . S$
      - $S \rightarrow a B S .$

- Items denote the idea of the viable prefix. E.g.
  - $S \rightarrow . a B S$: to be a viable prefix, terminal ‘a’ needs to be shifted
  - $S \rightarrow a . B S$: to be a viable prefix, a set of terminals need to be shifted and reduced to non-terminal ‘B’
States in the LR Parser

- Action of LL parser is governed by: (stack top symbol, input)
- For LR parser, more complex: (stack top state, input)
  - State: where we are in the current production at that point
  - State is denoted by a set of LR(0) items
- Why a set of LR(0) items?
  - There may be multiple candidate productions for the prefix. E.g.
    For grammar $S \rightarrow a \ b | a \ c$,
    $S \rightarrow a \ . \ b$ and $S \rightarrow a \ . \ c$ would be items in the same set
  - Even for one production, you may need multiple items expressing the same position, if the following symbol is a non-terminal. E.g.
    For grammar $S \rightarrow a \ B$, $B \rightarrow b$
    $S \rightarrow a \ . \ B$ and $B \rightarrow . \ b$ would be items in the same set
- LR parsers keep track of states alongside symbols in stack
### Parser Implementation in More Detail

- Each grammar symbol $X_i$ is associated with a state $S_i$.
- Contents of stack ($X_1X_2\ldots X_m$) is a viable prefix.
- Contents of stack + input ($X_1X_2\ldots X_ma_ia_n\ldots a_n$) is a right sentential form.
  - If the input string is a member of the language.
- Uses **state** at the top of stack and current input to index into parsing table to determine whether to shift or reduce.

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>---</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$X_1$</td>
</tr>
<tr>
<td>$S_{m-1}$</td>
<td>$X_{m-1}$</td>
</tr>
<tr>
<td>$S_m$</td>
<td>$X_m$</td>
</tr>
<tr>
<td>$S_{m+1}$</td>
<td>$X_{m+1}$</td>
</tr>
</tbody>
</table>

---

**LR Parsing program**

- **Table**
  - **Action**
  - **GoTo**

**Input string**

```
a_1 a_2 a_i a_n $
```
Parser Actions

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>S_m+1</th>
<th>X_{m+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_m</td>
<td>X_m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_{m-1}</td>
<td>X_{m-1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_1</td>
<td>X_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_0</td>
<td>---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

```

- **Shift**: 
  - S_m: A → X_m \cdot X_{m+1}
  - S_{m-1}: A → .X_m X_{m+1}
  - B → X_{m-1}.A

- **Reduce(1)**: 
  - S_{m+1}: A → X_m X_{m+1} \cdot
  - S_m: A → X_m \cdot X_{m+1}
  - B → X_{m-1}.A

- **(2)**: 
  - S_{m-1}: A → X_m X_{m+1}
  - S_1: X_1
  - S_0: ---

- **GOTO**: 
  - S_A: B → X_{m-1}.A
  - S_{m-1}: A → .X_m X_{m+1}
  - B → X_{m-1}.A

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>S_A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_{m-1}</td>
<td>X_{m-1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_1</td>
<td>X_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S_0</td>
<td>---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

```
Parser Actions

- Assume configuration = $S_0X_1S_1X_2S_2\ldots X_m S_m # a_i a_{i+1} \ldots a_n$

- Actions can be one of:
  1. Shift input $a_i$ and push new state $S$
     - New configuration = $S_0X_1S_1X_2S_2\ldots X_m S_m a_i S # a_{i+1} \ldots$
     - Where $\text{Action} [S_m, a_i] = s[S]$
  2. Reduce using Rule $R$ ($A \rightarrow \beta$) and push new state $S$
     - Let $k = |\beta|$, pop $2* k$ symbols and push $A$
     - New configuration = $S_0X_1S_1\ldots X_{m-k} S_{m-k} A S # a_i a_{i+1} \ldots$
     - Where $\text{Action} [S_{m-k}, A] = [S]$
  3. Accept – parsing is complete ($\text{Action} [S_m, a_i] = \text{accept}$)
  4. Error – report and stop ($\text{Action} [S_m, a_i] = \text{error}$)
Parse Table: Action and Goto

- **Action** \([S_m, a_i]\) can be one of:
  - **s[S]:** shift input symbol \(a_i\) and push state \(S\)
    (One item in \(S_m\) must be of the form \(A \rightarrow \alpha \cdot a_i \beta\))
  - **r[R]:** reduce using rule \(R\) on seeing input symbol \(a_i\)
    (One item in \(S_m\) must be \(R: A \rightarrow \alpha \cdot \), where \(a_i \in \text{Follow}(A)\))
    - Use GoTo \([S_{m-|\alpha|}, A]\) to figure out state to push with \(A\)
  - **Accept** (One item in \(S_m\) must be \(S' \rightarrow S \cdot\) where \(S\) is the original start symbol, and \(a_i\) must be \$\))
  - **Error** (Cannot shift, reduce, accept on symbol \(a_i\) in state \(S_m\))

- **GoTo** \([S_m, X_i]\) is \([S]\):
  - Next state to push when pushing nonterminal \(X_i\) from a reduction
    (At least one item in \(S_m\) must be of the form \(A \rightarrow \alpha \cdot X_i \beta\))
  - Similar to shifting input except now we are matching a nonterminal
### Grammar

1. \( S \rightarrow E \)
2. \( E \rightarrow E + T \)
3. \( E \rightarrow T \)
4. \( T \rightarrow id \)
5. \( T \rightarrow (E) \)

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>$) $</td>
</tr>
<tr>
<td>( E )</td>
<td>$) $</td>
</tr>
<tr>
<td>( T )</td>
<td>$) $</td>
</tr>
</tbody>
</table>

### ACTION

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S0 )</td>
<td></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S1 )</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S2 )</td>
<td>r3</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>( S3 )</td>
<td>r4</td>
<td></td>
<td></td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>( S4 )</td>
<td></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S5 )</td>
<td>s7</td>
<td></td>
<td></td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>( S6 )</td>
<td>r5</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>( S7 )</td>
<td></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S8 )</td>
<td>r2</td>
<td></td>
<td></td>
<td>r2</td>
<td>r2</td>
</tr>
</tbody>
</table>

### GOTO

<table>
<thead>
<tr>
<th></th>
<th>( E )</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S0 )</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( S1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S2 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S3 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S4 )</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( S5 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S6 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S7 )</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>( S8 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Grammar
1. \( S \rightarrow E \)
2. \( E \rightarrow E + T \)
3. \( E \rightarrow T \)
4. \( T \rightarrow id \)
5. \( T \rightarrow (E) \)

Non-terminal | Follow
--- | ---
\( S \) | $\$
\( E \) | $ + )$ $\$
\( T \) | $ + )$ $\$

ACTION

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>r3</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>s7</td>
<td></td>
<td>s6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>r2</td>
<td></td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
</tbody>
</table>

GOTO

<table>
<thead>
<tr>
<th>E</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>S5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>S8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parse Table in Action

- Example input string
  \[ \text{id} + \text{id} + \text{id} \]

- Parser actions

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>\text{id} + \text{id} + \text{id} $</td>
<td>Action[S0, id] (s3): Shift “id”, Push S3</td>
</tr>
<tr>
<td>S0 id S3</td>
<td>+ \text{id} + \text{id} $</td>
<td>Action[S3, +] (r4): Reduce rule 4 (T→id)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GoTo[S0, T] (2): Push S2</td>
</tr>
<tr>
<td>S0 T S2</td>
<td>+ \text{id} + \text{id} $</td>
<td>Action[S2, +] (r3): Reduce rule 3 (E→T)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GoTo[S0, E] (1): Push S1</td>
</tr>
<tr>
<td>S0 E S1</td>
<td>+ \text{id} + \text{id} $</td>
<td>Action[S1, +] (s7): Shift “+”, Push S7</td>
</tr>
<tr>
<td>S0 E S1 + S7</td>
<td>\text{id} + \text{id} $</td>
<td>Action[S7, id] (s7): Shift “id”, Push S3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>S0 E S1 + S7 T S8</td>
<td>+ \text{id} $</td>
<td>Action[S8, +] (r2): Reduce rule 2 (E→E+T)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GoTo[S0, E] (1): Push S1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Power Added to DFA by Stack

- LR parser is basically DFA+Stack (Pushdown Automaton)
- DFA: can only remember one state ( “dot” in current rule )
- DFA + Stack: remembers current state and all past states (“dots” in rules higher up in the tree waiting for non-terminals)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>id + id $</td>
<td>s3</td>
</tr>
<tr>
<td>S0 id S3</td>
<td>+ id $</td>
<td>r4, goto[S0, T]</td>
</tr>
<tr>
<td>S0 T S2</td>
<td>+ id $</td>
<td>r3, goto[S0, E]</td>
</tr>
<tr>
<td>S0 E S1</td>
<td>+ id $</td>
<td>s7</td>
</tr>
<tr>
<td>S0 E S1 + S7</td>
<td>id $</td>
<td>s3</td>
</tr>
<tr>
<td>S0 E S1 + S7 id S3</td>
<td>$</td>
<td>r4, goto[S7, T]</td>
</tr>
<tr>
<td>S0 E S1 + S7 T S8</td>
<td>$</td>
<td>r2, goto[S0, E]</td>
</tr>
<tr>
<td>S0 E S1</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Remember the following CFG for the language \{ [i \, i] \mid i \geq 1 \}\? 
S \rightarrow [ S ] \mid [ ]

Regular grammars (or DFAs) could not recognize language because the state machine had to “count”

LR parsers can use stack to count by pushing as many states as there are [ symbols in the input string

Q: Is this language LL(1)?

Yes. After left-factoring.
S \rightarrow [ S', S' \rightarrow S ] \mid ]

Now stack counts ] symbols.

Same pushdown automaton but different usage
LR Parse Table Construction

- Must be able to decide on action from:
  - State at the top of stack
  - Next k input symbols (In practice, k = 1 is often sufficient)

- To construct LR parse table from grammar
  - Two phases
    - Build deterministic finite state automaton to go from state to state
    - Express DFA using Action and GoTo tables

- State: Where we are currently in the structure of the grammar
  - Expressed as a set of LR(0) items
  - Each item expresses position in terms of the RHS of a rule
Construction of LR States

1. Create augmented grammar $G'$ for $G$
   - Given $G$: $S \rightarrow \alpha \mid \beta$, create $G'$: $S' \rightarrow S$ $S \rightarrow \alpha \mid \beta$
   - Creates a single rule $S' \rightarrow S$ that when reduced, signals acceptance
2. Create first state by performing a closure on initial item $S' \rightarrow . S$
   - Closure($I$): computes set of items expressing the same position as $I$
   - Closure($\{S' \rightarrow . S\}$) = $\{S' \rightarrow . S, S \rightarrow . \alpha, S \rightarrow . \beta\}$
3. Create additional states by performing a goto on each symbol
   - Goto($I, X$): creates state that can be reached by advancing $X$
   - If $\alpha$ was single symbol, the following new state would be created:
     Goto($\{S' \rightarrow . S, S \rightarrow . \alpha, S \rightarrow . \beta\}, \alpha$) =
     Closure($\{S \rightarrow \alpha .\}$) = $\{S \rightarrow \alpha .\}$
4. Repeatedly perform gotos until there are no more states to add
Closure Function

- Closure(I) where I is a set of items
  - Returns the state (set of items) that express the same position as I
  - Items in I are called kernel items
  - Rest of items in closure(I) are called non-kernel items

- Let N be a non-terminal
  - If dot is in front of N, then add each production for that N and put dot at the beginning of the RHS
    - A \rightarrow \alpha \cdot \beta \text{ is in I} \quad \text{; we expect to see a string derived from B}
    - B \rightarrow \cdot \gamma \text{ is added to the closure, where } B \rightarrow \gamma \text{ is a production}
    - Apply rule until nothing is added

- Given
  
  \begin{align*}
  & S \rightarrow E \\
  & E \rightarrow E + T \\
  & E \rightarrow T \\
  & T \rightarrow \text{id} \mid ( E )
  \end{align*}

  \text{Closure(\{ S \rightarrow \cdot E \}) = \{ S \rightarrow \cdot E, E \rightarrow \cdot E + T, E \rightarrow \cdot T, T \rightarrow \cdot \text{id}, T \rightarrow \cdot ( E ) \} }
Kernel and Non-kernel Items

- Two kinds of items
  - Kernel items
    - Items that act as “seed” items when creating a state
    - What items act as seed items when states are created?
      - Initial state: $S' \rightarrow . S$
      - Additional states: from goto(I, X) so has X at left of dot
      - Besides $S' \rightarrow . S$, all kernel items have dot in middle of RHS
  - Non-kernel items
    - Items added during the closure of kernel items
    - All non-kernel items have dot at the beginning of RHS
Goto Function

- Goto (I, X) where I is a set of items and X is a symbol
  - Returns state (set of items) that can be reached by advancing X
  - For each $A \rightarrow \alpha \cdot X \beta$ in I,
    \[
    \text{Closure}(A \rightarrow \alpha X \beta) \text{ is added to } \text{goto}(I, X)
    \]
  - X can be a terminal or non-terminal
    - Terminal if obtained from input string by shifting
    - Non-terminal if obtained from reduction
- Example
  - Goto($\{T \rightarrow .(E)\}$, () = closure($\{T \rightarrow .(E)\}$)
- Generates next state after matching a terminal or non-terminal
Construction of DFA

- Algorithm to compute set C (set of all states in DFA)
  ```
  void items (G') {
      C = {closure({S→ . S})}  // Add initial state to C
      repeat
          for (each state I in C)
              for (each grammar symbol X)
                  if (goto(I, X) is not empty and not in C)
                      add goto(I, X) to C
              until no new states are added to C
  }
  ```
- All new states are added through goto(I, X)
  - States transitions are done on symbol X
Example: \[ S \rightarrow E \\
E \rightarrow E + T | T \\
T \rightarrow \text{id} | ( E ) \]

- \[ S_0 = \text{closure} (\{S \rightarrow . E\}) = \{S \rightarrow . E, E \rightarrow . E + T, E \rightarrow . T, T \rightarrow . \text{id}, T \rightarrow . ( E )\} \]
- \[ \text{goto}(S_0, E) = \text{closure} (\{S \rightarrow E ., S \rightarrow E . + T\}) \]
  \[ S_1 = \{S \rightarrow E . , S \rightarrow E . + T\} \]
- \[ \text{goto}(S_0, T) = \text{closure} (\{E \rightarrow T .\}) \]
  \[ S_2 = \{E \rightarrow T . \} \]
- \[ \text{goto}(S_0, \text{id}) = \text{closure} (\{T \rightarrow \text{id} .\}) \]
  \[ S_3 = \{T \rightarrow \text{id} . \} \]

- ....
- \[ S_8 = \ldots \]
DFA for the previous grammar
(* are closures applied to kernel items *)

S₀
- * S → E
- E → E . E + T
- E → . T
- T → . Id
- T → . (E)

S₁
- * S → E .
- * E → E . + T

S₂
- * E → T .

S₃
- * T → id .

S₄
- * T → ( . E)
- E → E . E + T
- E → . T
- T → . Id
- T → . (E)

S₅
- * T → (E .)
- * E → E . + T

S₆
- * T → (E .)

S₇
- * E → E . + T

S₈
Building Parse Table from DFA

• ACTION [state, terminal symbol]

• GOTO [state, non-terminal symbol]

ACTION:

1. If \([A \rightarrow \alpha \cdot a \beta] \) is in \(S_i\) and \(\text{goto}(S_i, a) = S_j\), where “a” is a terminal then \(\text{ACTION}[S_i, a] = \text{shift} \, j \, (s_j)\)

2. If \([A \rightarrow \alpha \cdot] \) is in \(S_i\) and \(A \rightarrow \alpha\) is rule number \(j\) then \(\text{ACTION}[S_i, a] = \text{reduce} \, j \, (r_j)\), for all \(a \in \text{Follow}(A)\)

3. If \([S' \rightarrow S_0 \cdot] \) is in \(S_i\) then \(\text{ACTION}[S_i, \$] = \text{accept}\)

   If no conflicts among 1 and 2

   then it is said that this parser is able to parse the given grammar

GOTO

1. if \(\text{goto}(S_i, A) = S_j\) then \(\text{GOTO}[S_i, A] = j\)

All entries not filled are rejects
Grammar

1. \( S \rightarrow E \)
2. \( E \rightarrow E + T \)
3. \( E \rightarrow T \)
4. \( T \rightarrow \text{id} \)
5. \( T \rightarrow (E) \)

### Non-terminal Follow

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>$</td>
</tr>
<tr>
<td>( E )</td>
<td>+ ) $</td>
</tr>
<tr>
<td>( T )</td>
<td>+ ) $</td>
</tr>
</tbody>
</table>

### ACTION

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### GOTO

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parsers in the LR Family

Pitt, CS 2210
Types of LR Parsers

- **SLR** – simple LR (what we saw so far was SLR(1))
  - Easiest to implement
  - Not as powerful

- **Canonical LR**
  - Most powerful
  - Expensive to implement

- **LALR**
  - Look ahead LR
  - In between the 2 previous ones in power and overhead

Overall parsing algorithm is the same – table is different
Consider the grammar $G$

$$S \rightarrow A\ b\ c\ |\ B\ b\ d$$

$A \rightarrow a$

$B \rightarrow a$

$b \in \text{Follow}(A)$ and also $b \in \text{Follow}(B)$

What is reduced when “a b” is seen? reduce to A or B?

- Reduce-reduce conflict

$G$ is not SLR(1) but SLR(2)

- We need 2 symbols of look ahead to look past b:
  - b c – reduce to A
  - b d – reduce to B

- Possible to extend SLR(1) to k symbols of look ahead – allows larger class of CFGs to be parsed
SLR(k)

- Extend SLR(1) definition to SLR(k) as follows

  let $\alpha, \beta \in V^*$

  - $\text{First}_k(\alpha) = \{ x \in V_T^* | (\alpha \rightarrow^* x\beta \text{ and } |x| \leq k) \}$ gives
    - all terminal strings of size $\leq k$ derivable from $\alpha$
    - all $k$-symbol terminal prefixes of strings derivable from $\alpha$

  - $\text{Follow}_k(B) = \{ w \in V_T^* | S \rightarrow^* \alpha B \gamma \text{ and } w \in \text{First}_k(\gamma) \}$ gives
    - all $k$ symbol terminal strings that can follow $B$ in some derivation
    - all shorter terminal strings that can follow $B$ in some derivation
Parse Table

Let S be a state and lookahead b ∈ V_{T}^{*} such that |b|≤k

1. If A → α. ∈ S and b ∈ \text{Follow}_{k}(A) then
   - Action(S,b) – reduce using production A → α,

2. If D → α.a γ ∈ S and a ∈ V_{T} and b ∈ \text{First}_{k}(a γ \text{Follow}_{k}(D))
   - Action(S,b) = shift “a” and push state goto(S,a)

For k =1, this definition reduces to SLR(1)
   Reduce: Trivially true
   Shift: \text{First}_{1}(a γ \text{Follow}_{1}(D)) = \{a\}
SLR(k)

Consider

\[ S \to A \, b^{k-1} \, c \mid B \, b^{k-1} \, d \]
\[ A \to a \]
\[ B \to a \]

SLR(k) not SLR(k-1)
- cannot decide what to reduce,
- reduce a to A or B depends on the next k symbols
  \[ b^{k-1} \, c \mid b^{k-1} \, d \]
Non SLR(k)

Consider another Grammar G

\[ S \rightarrow j A j \mid A m \mid a j \]
\[ A \rightarrow a \]

Follow(A) = \{j, m\}
State S1: [A \rightarrow a.] – reduce using this production (on j or m)
[S \rightarrow a.j] – shift j \rightarrow shift-reduce conflict \rightarrow not SLR(1)

? SLR(k)?
For reducing A \rightarrow a.: Follow\(_k\)(A) = First\(_k\)(j) + First\(_k\)(m) = \{j, m\},
For shifting S \rightarrow a.j: First\(_k\)(jFollow\(_k\)(S)) = \{j\} \text{ so not SLR(k) for any k}!!!
Why?

- Look ahead is too crude
  - In S1, if $A \rightarrow a$ is reduced then $\{m\}$ is the only possible symbol that can be seen – the only valid look ahead
  - Fact that $\{j\}$ can follow $A$ in another context is irrelevant

- Want to compare look ahead in a state to those symbols that might actually occur in the context represented by the state.

- Done in Canonical LR !!!
  - Determine look ahead appropriate to the context of the state
  - States contains a look ahead – will be used only for reductions
Follow(A)={a,b,c}

SLR(1)

LALR(1)

state merging

LR(1)

state splitting
Constructing Canonical LR

- Problem: Follow set in SLR is not precise enough
  - Follow set ignores context where reduction for item occurs
- Solution: Define a more precise follow set
  - For each item, encode a more precise follow set according to context
  - Use more precise follow set when deciding whether to reduce item
- LR(1) item: LR item with one lookahead
  - \([A \rightarrow \alpha.\beta, a]\) where \(A \rightarrow \alpha\beta\) is a production and \(a\) is a terminal or $\$
    - Meaning: Only terminal \(a\) can follow \(A\) in this context
    - Interpretation: After \(\beta\) is shifted and eventually we reach \([A \rightarrow \alpha\beta., a]\), only reduce \(A\) if lookahead matches terminal \(a\)
  - Second lookahead component will always be a subset of \(\text{Follow}(A)\)
Constructing Canonical LR

- Essentially the same as LR(0) items only adding lookahead
  - Modify closure and goto function

- Changes for closure
  - \([A \rightarrow \alpha. B\beta, a]\) and \(B \rightarrow \delta\) then
    \([B \rightarrow \delta, c]\) where \(c \in \text{First}(\beta a)\)

- Changes for goto function
  - Carry over lookahead
    \([A \rightarrow \alpha X\beta, a] \in I\) then goto \((I, X) = [A \rightarrow \alpha X. \beta, a]\)
Example

- Grammar
  \[
  S' \rightarrow S \\
  S \rightarrow CC \\
  C \rightarrow eC \mid d
  \]

- S0: closure(S' \rightarrow S, $)
  
  \[
  [S' \rightarrow S, $] \\
  [S \rightarrow CC, $] \quad \text{first}(\varepsilon$)=$\{\}$ \\
  [C \rightarrow eC, e/d] \quad \text{first}(C$)=$\{e,d\}$ \\
  [C \rightarrow d, e/d] \quad \text{first}(C$)=$\{e,d\}$
  \]

- S1: goto(S0, S) = closure(S' \rightarrow S, $)
  
  \[
  [S' \rightarrow S, $] \\
  [S \rightarrow CC, $] \quad \text{first}(\varepsilon$)=$\{\}$ \\
  [C \rightarrow eC, e/d] \quad \text{first}(C$)=$\{e,d\}$ \\
  [C \rightarrow d, e/d] \quad \text{first}(C$)=$\{e,d\}$
  \]

- S2: goto(S0, C) = closure(S \rightarrow C.C, $)
  
  \[
  [S \rightarrow C.C, $] \\
  [C \rightarrow eC, $] \quad \text{first}(\varepsilon$)=$\{\}$ \\
  [C \rightarrow d, $] \quad \text{first}(\varepsilon$)=$\{\}$
<table>
<thead>
<tr>
<th>Step</th>
<th>Transition</th>
<th>Action</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>goto(S0,e)</td>
<td>closure(C\rightarrow e.C, e/d)</td>
<td>[C \rightarrow e.C, e/d]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[C \rightarrow e.C, e/d]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>first(\epsilon e/d) = {e,d}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[C \rightarrow e.C, e/d]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>first(\epsilon e/d) = {e,d}</td>
</tr>
<tr>
<td>S4</td>
<td>goto(S0, d)</td>
<td>closure(C\rightarrow d., e/d)</td>
<td>[C \rightarrow d., e/d]</td>
</tr>
<tr>
<td>S5</td>
<td>goto(S2, C)</td>
<td>closure(S\rightarrow CC., $)</td>
<td>[S \rightarrow CC., $]</td>
</tr>
<tr>
<td>S6</td>
<td>goto(S2, e)</td>
<td>closure(C\rightarrow e.C, $)</td>
<td>[C \rightarrow e.C, $]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>first(\epsilon$) = {$}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[C \rightarrow e.C, $]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>first(\epsilon$) = {$}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[C \rightarrow d, $]</td>
</tr>
<tr>
<td>S7</td>
<td>goto(S2, d)</td>
<td>closure(C\rightarrow d., $)</td>
<td>[C \rightarrow d., $]</td>
</tr>
<tr>
<td>S8</td>
<td>goto(S3, C)</td>
<td>closure(C\rightarrow eC., e/d)</td>
<td>[C \rightarrow eC., e/d]</td>
</tr>
<tr>
<td>S9</td>
<td>goto(S6, C)</td>
<td>closure(C\rightarrow eC., $)</td>
<td>[C \rightarrow eC., $]</td>
</tr>
</tbody>
</table>
Note S3, S6 are same except for lookahead (also true for S4, S7 and S8, S9)
In SLR(1) – one state represents both
Constructing Canonical LR Parse Table

- Shifting: same as before
- Reducing:
  - Don’t use follow set (too coarse grain)
  - Reduce only if input matches lookahead for item
- Action and GOTO
  1. if \([A \rightarrow \alpha \bullet a \beta, b] \in S_i\) and goto(S_i, a) = S_j,
     \[\text{Action}[I,a] = s[S_j] - \text{shift and goto state } j \text{ if input matches } a\]
     \textit{Note: same as SLR}
  1. if \([A \rightarrow \alpha \bullet, a] \in S_i\)
     \[\text{Action}[I,a] = r[R] - \text{reduce } R: A \rightarrow \alpha \text{ if input matches } a\]
     \textit{Note: for SLR, reduced if input matches Follow(A)}
  2. if \([S' \rightarrow S., \$] \in S_i\),
     \[\text{Action}[i,\$] = \text{accept}\]
Revisit SLR and LR

\[ S \rightarrow aEa \mid bEb \mid aFb \mid bFa \]

\[ E \rightarrow e \]

\[ F \rightarrow e \]

SLR: reduce/reduce conflict

Follow(\(E,F\)) = \{\(a, b\)\}

\[ aea \Rightarrow E \rightarrow e \quad \text{and} \quad bea \Rightarrow F \rightarrow e \]

Is this LR(1)? Will not have a conflict because states will be split to take into account this context

\[ E \text{ if followed by } a/b \text{ preceded by } a/b, \text{ respectively} \]

\[ F \text{ if followed by } a/b \text{ preceded by } b/a, \text{ respectively} \]
SLR: Follow(E) = Follow(F) = \{a,b\}

LR: Follow sets more precise
SLR(1) and LR(1)

- LR(1) more powerful than SLR(1) – can parse more grammars
- But LR(1) may end up with many more states than SLR(1)
  - One LR(0) item may split up to many LR(1) items
    (As many as all combinations of lookahead possible – potentially powerset of entire alphabet)

- LALR(1) – compromise between LR(1) and SLR(1)
  - Constructed by merging LR(1) states with the same core
    - Ends up with same number of states as SLR(1)
    - But items still retain some lookahead info – still better than SLR(1)
  - Used in practice because most programming language syntactic structures can be represented by LALR (not true for SLR)
A \rightarrow B.

Follow(A) = \{a, b, c\}

\text{state merging}

LALR(1)

A \rightarrow B.,a/b

A \rightarrow B.,b/c

\text{state splitting}

LR(1)
Example

- **Grammar**
  
  \[ S' \rightarrow S \]
  
  \[ S \rightarrow CC \]
  
  \[ C \rightarrow eC \mid d \]

- **S3**: goto(S0,e) = closure(C→e.C, e/d)
  
  [C → e.C, e/d]
  
  [C → e.C, e/d]
  
  [C → e.C, e/d]

- **S4**: goto(S0, d) = closure(C→d., e/d)
  
  [C → d., e/d]

- **S5**: goto(S2, e) = closure(C→e.C, $)
  
  [C → e.C, $]

- **S6**: goto(S2, e) = closure(C→e.C, $)
  
  [C → e.C, $]

- **S7**: goto(S2, d) = closure(C→d., $)
  
  [C → d., $]

- **S8**: goto(S3, C) = closure(C→eC., e/d)
  
  [C → eC., e/d]

- **S9**: goto(S6, C) = closure(C→eC., $)
  
  [C → eC., $]
Note S3 and S6 are the same except for look ahead (same for S4 and S7)
In SLR(1) – one state represents both
Merging states

- Can merge S3 and S6

<table>
<thead>
<tr>
<th>S3: goto(S0,e)=closure(C→e.C, e/d)</th>
<th>S6: goto(S2,e)=closure(C→e.C, $)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[C →e.C, e/d]</td>
<td>[C →e.C, $]</td>
</tr>
<tr>
<td>[C →eC, e/d]</td>
<td>[C →eC, $]</td>
</tr>
<tr>
<td>[C →d, e/d]</td>
<td>[C →d, $]</td>
</tr>
</tbody>
</table>

- Similarly
  - S47: [C→d., e/d/$]
  - S89: [C→eC., e/d$/]
Effects of Merging

1. Detection of errors may be delayed
   - On error, LALR parsers will not perform shifts beyond an LR parser but may perform more reductions before finding error
   - Example: 
     \[ S' \rightarrow S \quad S \rightarrow CC \]
     \[ C \rightarrow eC \mid d \]
     and input string eed$
     
     - Canonical LR: Parse Stack S0 e S3 e S3 d S4
       State S4 on $ input = error S4: \{C \rightarrow d., e/d\}
     
     - LALR:
       \begin{align*}
       \text{stack: } & S0 \ e \ S36 \ e \ S36 \ d \ S47 & \rightarrow \text{state } S47 \text{ input }, \text{reduce } C \rightarrow d \\
       \text{stack: } & S0e \ S36 \ e \ S36 \ C \ S89 & \rightarrow \text{reduce } C \rightarrow eC \\
       \text{stack: } & S0 \ e \ S36 \ C \ S89 & \rightarrow \text{reduce } C \rightarrow eC \\
       \text{stack: } & S0 \ C \ S2 & \rightarrow \text{state S2 on input }, \text{error}
       \end{align*}
Effects of Merging

2. Merging of states can introduce conflicts
   - cannot introduce shift-reduce conflicts
   - can introduce reduce-reduce conflicts

- Shift-reduce conflicts
  Suppose $S_{ij}: [A \xrightarrow{\alpha}, a]$ reduce on input $a$
  $[B \xrightarrow{\beta.a\delta}, b]$ shift on input $a$
  formed by merging $S_i$ and $S_j$
  Cores are the same for $S_i$, $S_j$ and one of them must contain
  $[A \xrightarrow{\alpha}, a]$ and $[B \xrightarrow{\beta.a\delta}, b]$
  $\Rightarrow$ shift-reduce conflicts were already present in either $S_i$ and $S_j$ (or both) and not newly introduced by merging
Reduce-reduce Conflicts

\[ S \rightarrow aEa \mid bEb \mid aFb \mid bFa \]
\[ E \rightarrow e \]
\[ F \rightarrow e \]

S3: \[ [E \rightarrow e., a] \]
\[ [F \rightarrow e., b] \]

S4: \[ [E \rightarrow e., b] \]
\[ [F \rightarrow e., a] \]

Merging S34: \[ [E \rightarrow e., a/b] \]
\[ [F \rightarrow e., a/b] \]

- both reductions are called on inputs a and b, i.e. reduce-reduce conflict
Non SLR(k) but LALR(1)

Let’s consider the Non SLR(k) Grammar G again

- $S \rightarrow j\ A\ j \mid A\ m \mid a\ j$
- $A \rightarrow a$

Follow($A$) = {j, m}

State S1: 
- [A$\rightarrow$ a.] – reduce using this production (on j or m)
- [S$\rightarrow$ a.j] – shift j \(\rightarrow\) not SLR(1)

But LALR(1)

For reducing A$\rightarrow$a.: lookahead must be {m}
For shifting S$\rightarrow$a.j: First$_k$(jFollow$_k$(S)) = {j} \(\rightarrow\) so LALR(1)
Construction on LALR Parser

One solution:
- Construct LR(1) states
- Merge states with same core
- If no conflicts, you have a LALR parser

Inefficient because of building LR(1) items are expensive in time and space (then what is the point of using LALR?)

Efficient construction of LALR parsers
- Avoids initial construction of LR(1) states
- Merges states on-the-fly (step-by-step merging)
  - States are created as in LR(1)
  - On state creation, immediately merge if there is an opportunity
Compaction of LALR Parse Table

- A typical language grammar with 50-100 terminals and 100 productions may have an LALR parser with several hundred states and thousands of action entries

- Often multiple rows of table are identical so share the rows
  - Make states point to an array of unique rows

- Often most entries in a row are empty
  - Instead of an actual row array, use row lists of (input, action) pairs

- Slows access to the table but can reduce memory footprint
Error Recovery

- Error recovery: How does the parser uncover multiple errors?
- Error detected when parser consults parsing table and hits an empty entry
  - Compared to LR, SLR and LALR parser may go through several reductions before detecting an error
    - Due to more coarse-grained use of lookahead
    - But never shifts beyond an erroneous symbol
- Simple error recovery (by discarding offending code sequence)
  1. Decide on non-terminal A: candidate for discarding
    - Typically an expression, statement, or block
  2. Continue to scan down the stack until a state S with a goto on a particular non-terminal A is found
  3. Discard input symbols until a symbol ‘a’ is found that can follow A
    - E.g. if A is a statement then ‘a’ would be ‘;’
  4. Push state Goto[a,A] on stack and continue parsing
Using Automatic Tools
-- YACC

Pitt, CS 2210
Using a Parser Generator

YACC is an LALR(1) parser generator

- YACC: Yet Another Compiler-Compiler

YACC constructs an LALR(1) table and reports an error when a table entry is multiply defined

- A shift and a reduce – reports shift/reduce conflict
- Multiple reduces – reports reduce/reduce conflict
- Most conflicts are due to ambiguous grammars
- Must resolve conflicts
  - By specifying associativity or precedence rules
  - By modifying the grammar
  - YACC outputs detail about where the conflict occurred (by default, in the file “y.output”)
Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else

\[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER} \]

will have DFA state containing

- \[ [S \rightarrow \text{if } E \text{ then } S, \text{ else}] \]
- \[ [S \rightarrow \text{if } E \text{ then } S, \text{ else } S, \text{ else}] \]

so on ‘else’ we can shift or reduce

- Default (YACC, bison, etc.) behavior is to shift
  - Default behavior is the correct one in this case
  - Better not to rely on this and remove ambiguity
More Shift/Reduce Conflicts

Consider the ambiguous grammar

\[ E \rightarrow E + E \mid E^* E \mid \text{int} \]

we will have the states containing

\[ [E \rightarrow E^* \cdot E, +/*] \quad [E \rightarrow E^* E \cdot , +/*] \]
\[ [E \rightarrow \cdot E + E, +/*] \quad [E \rightarrow E . +E, +/*] \]

Again we have a shift/reduce conflict on input +

- In this case, we need to reduce (* is higher than +)
- Easy (better) solution: declare precedence rules for * and +
- Hard solution: rewrite grammar to be unambiguous
More Shift/Reduce Conflicts

- Declaring precedence and associativity in YACC
  
  %left ‘+’ ‘-’
  %left ‘*’ ‘/’

  ➢ Interpretation:
  - +, -, *, / are left associative
  - +, - have lower precedence compared to *, /
    (associativity declarations are in the order of increasing precedence)
  - Precedence of a candidate rule for reduction is the precedence of the last terminal in that rule (e.g. For ‘E→ E+E .’, level is same as ‘+’)

  ➢ Resolve shift/reduce conflict with a shift if:
  - No precedence declared for either rule or terminal
  - Input terminal has higher precedence than the rule
  - The precedence levels are the same and right associative
Use Precedence to Solve S/R Conflict

\[ E \rightarrow E \cdot E, +/* \] \quad \[ E \rightarrow E \cdot E, +/* \]
\[ E \rightarrow E * E \cdot , +/* \] \[ E \rightarrow E * E \cdot , +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]

we will choose reduce because precedence of rule

\[ E \rightarrow E * E \] is higher than that of terminal +

\[ E \rightarrow E + E \cdot E, +/* \] \[ E \rightarrow E + E \cdot , +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]
\[ E \rightarrow E * E \cdot E + E, +/* \] \[ E \rightarrow E + E, +/* \]

we will choose reduce because \( E \rightarrow E + E \) and + have the same precedence and + is left-associative
Back to our dangling else example

\[
[S \rightarrow \text{if } E \text{ then } S, \text{ else}]
\]

\[
[S \rightarrow \text{if } E \text{ then } S \text{ else } S, \text{ else}]
\]

- Can eliminate conflict by declaring ‘else’ with higher precedence than ‘then’

- But this looks much less intuitive compared to arithmetic operator precedence

- Best to avoid overuse of precedence declarations that do not enhance the readability of your code
Reduce/Reduce Conflicts

- Usually due to ambiguity in the grammar

- Example: a sequence of identifiers

  \[ S \rightarrow \varepsilon \mid \text{id} \mid \text{id} \ S \]

  There are two parse trees for the string ‘id’

  \[ S \rightarrow \text{id} \]
  \[ S \rightarrow \text{id} \ S \rightarrow \text{id} \]

  How does this confuse the parser?
Reduce/Reduce Conflicts

Consider the states

\[
\begin{align*}
[S' \rightarrow ., S, \$] & \quad [S \rightarrow \text{id}. , \$,] \\
[S \rightarrow ., \$] & \quad [S \rightarrow \text{id}.S, \$] \\
[S \rightarrow . \text{id}, \$] & \quad [S \rightarrow ., \$] \\
[S \rightarrow . \text{id} S, \$] & \quad [S \rightarrow . \text{id}, \$] \\
[S \rightarrow . \text{id} S, \$] & \quad [S \rightarrow . \text{id} S, \$]
\end{align*}
\]

Reduce/reduce conflict on input “id$”

\[
S' \rightarrow S \rightarrow \text{id} \\
S' \rightarrow S \rightarrow \text{id} S \rightarrow \text{id}
\]

Better rewrite the grammar: \( S \rightarrow \varepsilon \mid \text{id} S \)
Semantic Actions

- Semantic actions are implemented for LR parsing
  - keep attributes on the semantic stack – parallel to the parse stack
    - on shift a, push attribute for a on semantic stack
    - on reduce X → α
      - pop attributes for α
      - compute attribute for X
      - push it on the semantic stack

- Creating an AST
  - Bottom up
    - Create leaf node from attribute values of token(s) in RHS
    - Create internal node from subtree(s) passed on from RHS
Performing Semantic Actions

- Compute the value

  \[ E \rightarrow T + E_1 \]  
  \[ \text{\{E.val = T.val + E1.val\}} \]

  \[ T \rightarrow \text{int} \ast T_1 \]  
  \[ \text{\{T.val = int.val * T1.val\}} \]

- Consider the parsing of the string \(3 \ast 5 + 8\)

Recall: creating the AST

- \[ E \rightarrow \text{int} \]  
  \[ \text{E.ast = mkleaf(int.lexval)} \]

- \[ E \rightarrow E_1 + E_2 \]  
  \[ \text{E.ast = mktree(plus, E1.ast, E2.ast)} \]

- \[ E \rightarrow (E_1) \]  
  \[ \text{E.ast = E1.ast} \]

- A bottom-up evaluation of the ast attribute:

  \[ E.ast = \text{mktree(plus, mkleaf(5),} \]

  \[ \text{mktree(plus, mkleaf(2), mkleaf(3))}) \]
Hierarch of Grammar Classes

Pitt, CS 2210
A Hierarchy of Grammar Classes

Unambiguous Grammars

LL(k)       LR(k)
LL(1)       LR(1)
LALR(1)     SLR
LL(0)       LR(0)

Ambiguous Grammars
LALR vs. LR Parsing (LALR < LR)

- LR(k) is strictly more powerful compared to LALR(k)
  - LALR merges and reduces number of states.

- Slightly unintuitive since unlike LL, LR since no formal definition exists on what is LALR
  - Definition by construction: if LALR parser has no conflicts
  - If there is a reduce-reduce conflict, is it because of merging?

- However, LALR(1) has become a standard for programming languages and for parser generators
  - YACC, Bison, etc.
  - Most PLs have an LALR(1) grammar
  - Reduce-reduce conflicts due to merging are rare (mostly due to ambiguity)
LL vs. LR Parsing (LL < LR)

- LL(k) parser, each expansion $A \rightarrow \alpha$ is decided on the basis of
  - Current non-terminal at the top of the stack
    - Which LHS to produce
  - $k$ terminals of lookahead at \textit{beginning} of RHS
    - Must guess which RHS by peeking at first few terminals of RHS

- LR(k) parser, each reduction $A \rightarrow \alpha \cdot$ is decided on the basis of
  - RHS at the top of the stack
    - Can postpone choice of RHS until entire RHS is seen
    - Common left factor is okay – waits until entire RHS is seen anyway
    - Left recursion is okay – does not impede forming RHS for reduction
  - $k$ terminals of lookahead \textit{beyond} RHS
    - Can decide on RHS after looking at entire RHS plus lookahead
LL vs. SLR Parsing (LL != SLR)

- Neither is strictly more powerful than the other
- Advantage of SLR: can delay decision until entire RHS seen
  - LL must decide RHS with a few symbols of lookahead
- Disadvantage of SLR: lookahead applied out of context
  - Consider grammar: $S \rightarrow Bb \mid Cc \mid aBc$, $B \rightarrow \varepsilon$, $C \rightarrow \varepsilon$
  - Initial state $S_0 = \{ S \rightarrow . Bb \mid . Cc \mid . aBc, B \rightarrow ., C \rightarrow . \}$
  - For SLR(1), reduce-reduce conflict on $B \rightarrow .$ and $C \rightarrow .$
    - $\text{Follow}(B) = \{ b, c \}$ and $\text{Follow}(C) = \{ c \}$
  - For LL(1), no conflict
    - $\text{First}(Bb) = \{ b \}$, $\text{First}(Cc) = \{ c \}$, $\text{First}(aBC) = \{ a \}$
- For the same reason, LL != LALR
LR(0) == LALR(0) == SLR(0) > LL(0)

- LR(0) == LALR(0) == SLR(0)
  - No lookahead for reducing
    - Lookahead components are meaningless (hence LR==LALR==SLR)
    - Must reduce regardless of Follow sets
    - If a state contains a reduce item, there can be no other reduce items or shift items for that state, or there will be a conflict
  - Makes grammars very restrictive. Not used very much.

- LL(0) < LR(0)
  - LL(0) can only have one RHS per non-terminal to avoid conflict
  - LR(0) can still have multiple RHSs per non-terminal
  - E.g. S → a | b is not LL(0) but is LR(0)
L(Rec. Descent) == L(GLR) == L(CFG)

- **L(Recursive Descent) == L(CFG)**
  - Can parse all CFGs by trial-and-error until input string match
  - Including ambiguous CFGs (accepts first encountered parse tree)
  - A general top-down parser for all CFGs can be constructed by using LL(k) parsing table, and falling back on recursive descent

- **Does that make top-down parsers superior to bottom-up?**
  - No. Same trial-and-error strategy can be employed for bottom-up
  - GLR (Generalized LR) parser: a parser for all CFGs that relies on an LR parsing table, but falls back on trial-and-error on conflict
  - L(GLR(k)) == L(CFG)

- Any LR table (e.g. SLR, LALR, Canonical LR) can be used
- GLR implementations: GNU Bison etc. (but not Yacc)
Notes on Parsing

- Parsing
  - A solid foundation: CFG
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficient compromise: LALR(1)
  - LALR(1) parser generators