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
\[
\begin{align*}
Z &\to b \ M \ b \\
M &\to ( \ L \mid a \\
L &\to M \ a ) \mid)
\end{align*}
\]

Considering string: \( w = b \ (a \ a ) \ b \)

The rightmost derivation of this parse tree:

\[
Z \Rightarrow b \ M \ b \Rightarrow b \ ( L \ b \Rightarrow b \ ( M \ a ) \ b \Rightarrow b \ ( a \ a ) \ b
\]

Bottom up parsing involves finding “handles” (RHSs) to reduce

\[
b \ (a \ a ) \ b \Rightarrow b \ ( M \ a ) \ b \Rightarrow b \ ( L \ b \Rightarrow b \ M \ b \Rightarrow Z
\]
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 )$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>b ( a a ) b $</td>
<td>shift</td>
</tr>
<tr>
<td>$ b</td>
<td>( a a ) b $</td>
<td>shift</td>
</tr>
<tr>
<td>$ b (</td>
<td>a a ) b $</td>
<td>shift</td>
</tr>
<tr>
<td>$ b ( a</td>
<td>a ) b $</td>
<td>reduce</td>
</tr>
<tr>
<td>$ b ( M</td>
<td>a ) b $</td>
<td>shift</td>
</tr>
<tr>
<td>$ b ( M a</td>
<td>) b $</td>
<td>shift</td>
</tr>
<tr>
<td>$ b ( M a )</td>
<td>b $</td>
<td>reduce</td>
</tr>
<tr>
<td>$ b ( L</td>
<td>b $</td>
<td>reduce</td>
</tr>
<tr>
<td>$ b M</td>
<td>b $</td>
<td>shift</td>
</tr>
<tr>
<td>$ b M b</td>
<td>$</td>
<td>reduce</td>
</tr>
<tr>
<td>$ Z</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Handle Always Occurs at Top of Stack

Grammar

<table>
<thead>
<tr>
<th>Production</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E+E</td>
<td>id₁ + id₂ * id₃</td>
<td>id₁</td>
</tr>
<tr>
<td>E → E*E</td>
<td>E + id₂ * id₃</td>
<td>id₂</td>
</tr>
<tr>
<td>E → (E)</td>
<td>E + E * id₃</td>
<td>id₃</td>
</tr>
<tr>
<td>E → id</td>
<td>E + E * E</td>
<td>E*E</td>
</tr>
</tbody>
</table>

#: location of handle

Left of #: completely reduced (except for handle), Right of #: unreduced

id₁ # + id₂ * id₃⇒ E # + id₂ * id₃⇒ E + # id₂ * id₃⇒ E + id₂ # * id₃
⇒ E + E # * id₃⇒ E + E * id₃ #⇒ E + E * E #⇒ E + E #⇒ 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)
Locating Handles

- If reverse of rightmost derivation is followed, the handle can always be found at the top of the parse stack.
- In below parse tree, numbers indicate production order in rightmost derivation.

- Rightmost derivation in reverse: 7 6 5 4 3 2 1
- Note for each reduction of handle:
  - Right side: contains unconsumed terminals
  - Left side: already digested parse tree
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: \text{int} * \text{int} + \text{int}

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

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

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

$$b ( a \# a ) b \Rightarrow b ( M \# a ) b \Rightarrow b ( L \# b \Rightarrow b M \# ) b \Rightarrow Z\#$$

- 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 (\(.\)) 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 ($X_1X_2\ldots X_m a_i \ldots a_n$) 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
Parser Actions

\[ X_{m+1} \rightarrow X_m \cdot X_{m+1} \]
\[ S_m : A \rightarrow X_m \cdot X_{m+1} \]
\[ S_{m-1} : A \rightarrow X_m \cdot X_{m+1} \]
\[ B \rightarrow X_{m-1} \cdot A \]

\[ X_{m+2} \rightarrow X_m \cdot X_{m+1} \]

Reduce(1)

\[ S_{m+1} : A \rightarrow X_m \cdot X_{m+1} \cdot X_{m+2} \]
\[ S_m : A \rightarrow X_m \cdot X_{m+1} \cdot X_{m+2} \]
\[ S_{m-1} : A \rightarrow X_m \cdot X_{m+1} \cdot X_{m+2} \]
\[ B \rightarrow X_{m-1} \cdot A \]

GOTO

\[ A \rightarrow X_{m-1} \cdot X_m \cdot X_{m+1} \cdot X_{m+2} \]
\[ S_A : B \rightarrow X_{m-1} \cdot X_m \cdot X_{m+1} \cdot X_{m+2} \]
\[ S_{m-1} : A \rightarrow X_m \cdot X_{m+1} \cdot X_{m+2} \]
\[ B \rightarrow X_{m-1} \cdot A \]

Shift

\[ X_m \rightarrow X_m \cdot X_{m+1} \]
\[ S_m : A \rightarrow X_m \cdot X_{m+1} \]
\[ S_{m-1} : A \rightarrow X_m \cdot X_{m+1} \]
\[ B \rightarrow X_{m-1} \cdot A \]
Parser Actions

1. **Assume configuration** = $S_0X_1S_1X_2S_2\ldots X_mS_m#a_ia_{i+1}\ldots a_n$

2. **Actions** can be one of:
   1. **Shift input** $a_i$ and push new state $S$
      - New configuration = $S_0X_1S_1X_2S_2\ldots X_mS_m a_i S # a_{i+1}\ldots$
      - Where 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 = $S_0X_1S_1\ldots X_{m-k}S_{m-k} A S # a_ia_{i+1}\ldots$
      - Where Action $[S_m, a_i] = r[R]$ and GoTo $[S_{m-k}, A] = [S]$
   3. **Accept** – parsing is complete (Action $[S_m, a_i] = \text{accept}$)
   4. **Error** – report and stop (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. $T \rightarrow \text{id}$
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>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></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>s3</td>
<td>s4</td>
<td></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>s3</td>
<td>s4</td>
<td></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>8</td>
</tr>
<tr>
<td>S8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Grammar
1. S → E
2. E → E + T
3. E → T
4. T → id
5. T → (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></td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>s7</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></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>s6</td>
<td></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>r2</td>
<td>r2</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>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Parse Table in Action

- **Example input string**

  \[ \text{id} + \text{id} + \text{id} \]

- **Parser actions**

## Parse Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>id + id + id $</td>
<td>Action[S0, id] (s3): Shift “id”, Push S3</td>
</tr>
<tr>
<td>S0 id S3</td>
<td>+ id + 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>+ id + 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>+ id + id $</td>
<td>Action[S1, +] (s7): Shift “+”, Push S7</td>
</tr>
<tr>
<td>S0 E S1 + S7</td>
<td>id + 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>+ 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>
Power Added to DFA by Stack

- Remember the following CFG for the language \{ [^i]^i \mid i \geq 1 \}?
  \[
  S \rightarrow [ S ] | [ ]
  \]
- 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.
    \[
    S \rightarrow [ S', S' \rightarrow S ] | ]
    \]
  - 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: \( S \rightarrow \alpha | \beta \), create G’: \( S' \rightarrow S \quad S \rightarrow \alpha | \beta \)
   - 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’→ . S\}) = {S’→ . S, S → . \alpha, S → . \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:
     - \( \text{Goto(\{S’→ . S, S → . \alpha, S → . \beta\}, \alpha) =} \)
     - \( \text{Closure(\{S → \alpha .\}) = \{S → \alpha .\}} \)

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 $\rightarrow$ $\alpha$. B $\beta$ is in I; we expect to see a string derived from B
    - B $\rightarrow$. $\gamma$ is added to the closure, where B $\rightarrow$ $\gamma$ is a production
    - Apply rule until nothing is added
  - Given
    - S $\rightarrow$ E
    - E $\rightarrow$ E + T
    - E $\rightarrow$ T
    - T $\rightarrow$ id | ( E )

Closure( { S $\rightarrow$. E } ) = { S $\rightarrow$. E, E $\rightarrow$. E + T, E $\rightarrow$. T, T $\rightarrow$. id, T $\rightarrow$. ( 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 $A \rightarrow \alpha . X \beta$ in I,
    - Closure($A \rightarrow \alpha X \beta$) is added to 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 *)
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 \([A \rightarrow \alpha \cdot \cdot]\) 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 \([S' \rightarrow S_0 \cdot \cdot]\) 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. \(\text{if } \text{goto}(S_i, A) = S_j\) 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. \( T \rightarrow (E) \)

<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></td>
<td>s7</td>
<td></td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td>r3</td>
<td></td>
<td></td>
<td>r3</td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td>r4</td>
<td></td>
<td></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></td>
<td>s7</td>
<td></td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>S6</td>
<td></td>
<td>r5</td>
<td></td>
<td></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></td>
<td>r2</td>
<td></td>
<td></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>8</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
Conflict

- Consider the grammar G
  \[ S \rightarrow A \ b\ c \mid B\ b\ d \]
  \[ A \rightarrow a \quad \text{b} \in \text{Follow}(A) \quad \text{and also} \quad b \in \text{Follow}(B) \]
  \[ B \rightarrow a \]

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 $\alpha, \beta \in V^*$
    - $\text{First}_k(\alpha) = \{ x \in V_T^* \mid (\alpha \rightarrow^* x \beta \text{ and } |x| \leq k) \}$ gives
      - All prefixes of strings derivable from $\alpha$, where size $\leq k$
      - Given $A \rightarrow \gamma . \alpha$, if $k$ lookahead symbols match $\text{First}_k(\alpha)$, good idea to continue down that path and go on shifting $\alpha$

- $\text{Follow}_k(A) = \{ w \in V_T^* \mid S \rightarrow^* \alpha A \gamma \text{ and } w \in \text{First}_k(\gamma) \}$ gives
  - All strings that can follow $A$ in some derivation, where size $\leq k$
  - Given $A \rightarrow \alpha .$, if $k$ lookahead symbols match $\text{Follow}_k(\alpha)$, good idea to reduce $\alpha$ to $A$
Parse Table

Let $S$ be a state and lookahead $b \in V_T^*$ such that $|b| \leq k$

1. If $A \rightarrow \alpha. \in S$ and $b \in \text{Follow}_k(A)$ then
   - Action$(S,b)$ – reduce using production $A \rightarrow \alpha$,

2. If $D \rightarrow \alpha.a \gamma \in S$ and $a \in V_T$ and $b \in \text{First}_k(a \gamma \text{Follow}_k(D))$
   - Action$(S,b)$ = shift “$a$” and push state $\text{goto}(S,a)$

For $k = 1$, this definition reduces to SLR(1)
- Reduce: Trivially true
- Shift: $\text{First}_1(a \gamma \text{Follow}_1(D)) = \{a\}$
SLR(k)

Consider

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

SLR(k) not SLR(k-1)

- cannot decide what to reduce,
- reduce a to A or B depends the next k symbols
  \[ b^{k-1} \, c \text{ or } b^{k-1} \, d \]
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→a.] – reduce using this production (on j or m)
[S→a.j] – shift j → shift-reduce conflict → not SLR(1)

? SLR(k)?

For reducing A→a.: Follow$_k$(A) = \{j$, m$\},
For shifting S→a.j: First$_k$(jFollow$_k$(S)) = \{j$\} 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)

X
→
Y
→
Z
→
A → B.
...
A → B.

LALR(1)

X
→
Y
→
Z
→
A → B.,a/b
...
A → B.,c

LR(1)

X
→
A → B.,a
Y
→
A → B.,b
Z
→
...
A → B.,c
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
  - \([A \rightarrow \alpha. B\beta, a] \text{ and } B \rightarrow \delta\) then
    \([B \rightarrow \delta, c] \text{ where } c \in \text{First}(\beta a)\)

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

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

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

- S1: goto(S0, S) = closure(S' \rightarrow S., $)
  \[ S' \rightarrow S., \$ \]

- S2: goto(S0, C) = closure(S \rightarrow C.C, $)
  \[ S \rightarrow C.C, \$ \]
  \[ C \rightarrow eC, \$ \] 
  \[ C \rightarrow d, \$ \] 
  \[ \text{first}(\varepsilon\$) = \{\$\} \]
  \[ \text{first}(\varepsilon\$) = \{\$\} \]
S3: $\text{goto}(S0, e) = \text{closure}(C \rightarrow eC, e/d)$

\begin{align*}
[C \rightarrow eC, e/d] \\
[C \rightarrow .eC, e/d] & \text{ first}(\varepsilon e/d) = \{e,d\} \\
[C \rightarrow .d, e/d] & \text{ first}(\varepsilon e/d) = \{e,d\}
\end{align*}

S4: $\text{goto}(S0, d) = \text{closure}(C \rightarrow .d., e/d)$

\begin{align*}
[C \rightarrow .d., e/d]
\end{align*}

S5: $\text{goto}(S2, C) = \text{closure}(S \rightarrow CC., $)

\begin{align*}
[S \rightarrow CC., $]
\end{align*}

S6: $\text{goto}(S2, e) = \text{closure}(C \rightarrow eC, $)

\begin{align*}
[C \rightarrow eC, $] \\
[C \rightarrow .eC, $] & \text{ first}(\varepsilon$) = \{$\}$ \\
[C \rightarrow .d, $] & \text{ first}(\varepsilon$) = \{$\}$
\end{align*}

S7: $\text{goto}(S2, d) = \text{closure}(C \rightarrow .d., $)

\begin{align*}
[C \rightarrow .d., $]
\end{align*}

S8: $\text{goto}(S3, C) = \text{closure}(C \rightarrow eC., e/d)$

\begin{align*}
[C \rightarrow eC., e/d]
\end{align*}

S9: $\text{goto}(S6, C) = \text{closure}(C \rightarrow eC., $)

\begin{align*}
[C \rightarrow eC., $]
\end{align*}
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 \([A \rightarrow \alpha \cdot a \beta, b] \in S_i\) and \(\text{goto}(S_i, a) = S_j\),
   \(\text{Action}[I,a] = s[S_j]\) – shift and goto state \(j\) if input matches a
   \(\text{Note: same as SLR}\)
2. if \([A \rightarrow \alpha \cdot, a] \in S_i\)
   \(\text{Action}[I,a] = r[R]\) – reduce \(R: A \rightarrow \alpha\) if input matches a
   \(\text{Note: for SLR, reduced if input matches } \text{Follow}(A)\)
Revisit SLR and LR

- $S \rightarrow aEa \mid bEb \mid aFb \mid bFa$
- $E \rightarrow e$
- $F \rightarrow e$

SLR(1): reduce/reduce conflict
- Since $\text{Follow}(E) = \text{Follow}(F) = \{a,b\}$

LR(1): no conflict
- $\text{Follow}(E) = \{a\}$ in the context of $S \rightarrow a.Ea$
- $\text{Follow}(F) = \{b\}$ in the context of $S \rightarrow a.Fb$
- **SLR:** $\text{Follow}(E) = \text{Follow}(F) = \{a, b\}$

- **LR:** Follow sets more precise

---

- $S \rightarrow a.Ea$
- $S \rightarrow a.Fb$
- $E \rightarrow e$
- $F \rightarrow e$

- $S \rightarrow b.Eb$
- $S \rightarrow b.Fa$
- $E \rightarrow e$
- $F \rightarrow e$

- $E \rightarrow e.$
- $F \rightarrow e.$

- $E \rightarrow e.$
- $F \rightarrow e.$

---

- $S \rightarrow .aEa$
- $S \rightarrow .bEb$
- $S \rightarrow .aFb$
- $S \rightarrow .bFa$

---

- $S \rightarrow b.Eb$
- $S \rightarrow b.Fa$
- $E \rightarrow e$
- $F \rightarrow e$

- $E \rightarrow e.$
- $F \rightarrow e.$

- $S \rightarrow a.e.$
- $S \rightarrow e.$
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)
  - Popular since most prog. languages are LALR (but not SLR)
Follow($A$) = \{a, b, c\}

SLR(1)

LALR(1)

state merging

LR(1)

state splitting
Example

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

- S3: goto(S0, e) = closure(C \rightarrow e.C, e/d)
  
  \[
  [C \rightarrow e.C, e/d] \\
  [C \rightarrow eC, e/d] \\
  [C \rightarrow d, e/d]
  \]

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

- S5: goto(S2, e) = closure(C \rightarrow e.C, $)
  
  \[
  [C \rightarrow e.C, $] \\
  [C \rightarrow eC, $] \\
  [C \rightarrow d, $]
  \]

- S6: goto(S2, e) = closure(C \rightarrow e.C, $)
  
  \[
  [C \rightarrow e.C, $] \\
  [C \rightarrow eC, $] \\
  [C \rightarrow d, $]
  \]

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

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

- S9: goto(S6, C) = closure(C \rightarrow eC., $)
  
  \[
  [C \rightarrow eC., $]
  \]
States S3, S4, S8 have the same core as S6, S7, S9, respectively
   ➢ All items are identical except for look ahead
Set of states that have been split when going from SLR(1) to LR(1)
   ➢ Merged back such that LALR(1) and SLR(1) have the same number of states
Merging states

- Merging S3 and S6

\[
\begin{align*}
S3: \text{goto}(S0,e) = \text{closure}(C \rightarrow &e.C, e/d) \\
&[C \rightarrow e.C, e/d] \\
&[C \rightarrow .eC, e/d] \\
&[C \rightarrow .d, e/d]
\end{align*}
\]

\[
\begin{align*}
S6: \text{goto}(S2,e) = \text{closure}(C \rightarrow &e.C, \$) \\
&[C \rightarrow e.C, \$] \\
&[C \rightarrow .eC, \$] \\
&[C \rightarrow .d, \$]
\end{align*}
\]

\[
\begin{align*}
S36: \quad & [C \rightarrow e.C, e/d/\$] \\
& [C \rightarrow .eC, e/d/\$] \\
& [C \rightarrow .d, e/d/\$]
\end{align*}
\]

- Similarly
  - S47: \([C \rightarrow .d, e/d/\$]
  - S89: \([C \rightarrow eC., e/d/\$] \)
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:
       stack: S0 e S36 e S36 d S47 \rightarrow state S47 input $, reduce C \rightarrow d
       stack: S0 e S36 e S36 C S89 \rightarrow reduce C \rightarrow eC
       stack: S0 e S36 C S89 \rightarrow reduce C \rightarrow eC
       stack: S0 C S2 \rightarrow state S2 on input $, error
Effects of Merging

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

Shift-reduce conflicts
Suppose \( S_{ij} \):
- \([A \rightarrow \alpha, a]\) reduce on input \(a\)
- \([B \rightarrow \beta.a\delta, b]\) 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
- \([A \rightarrow \alpha, a]\) and \([B \rightarrow \beta.a\delta, b]\)
- 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 → aEa | bEb | aFb | bFa
E → e
F → e

S3: [E → e, a] viable prefix ae
    [F → e, b]
S4: [E → e, b] viable prefix be
    [F → e, a]
Merging S34: [E → e, a/b]
    [F → e, a/b]

• Both reductions are possible 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)
\[S \rightarrow a.j, \, \$\]
\[S \rightarrow A.m, \, \$\]
\[S \rightarrow a.j, \, \$\]
\[A \rightarrow a, \, m\]

No shift-reduce conflict on j!

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 \text{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

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

will have DFA state containing

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

Shift-reduce conflict on ‘else’

- 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

\[ [E \rightarrow E* . E, +/*] \quad [E \rightarrow E*E . , +/] \]
\[ [E \rightarrow . E+E, +/*] \quad \Rightarrow \quad [E \rightarrow E . +E, +/] \]

\[ \ldots \quad \ldots \]

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

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

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

we will choose reduce because precedence of terminal * in rule \( E \rightarrow E^*E \) is higher than that of terminal +

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

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
  \[ S \rightarrow \varepsilon \mid id \mid id\ S \]
  There are two parse trees for the string ‘id’
  \[ S \rightarrow id \]
  \[ S \rightarrow id\ S \rightarrow id \]
  How does this confuse the parser?
Reduce/Reduce Conflicts

Consider the states

\[ S' \rightarrow S, \] $] \quad [S \rightarrow \text{id.}, $]$
\[ S \rightarrow , \] $] \quad [S \rightarrow \text{id.S, } $]$
\[ S \rightarrow \text{id, } $] \Rightarrow \quad [S \rightarrow , \] $]$
\[ S \rightarrow \text{id S, } $] \quad [S \rightarrow \text{id, } $]$
\[ S \rightarrow \text{id S, } $]

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 $X \rightarrow \alpha$
      - pop attributes for $\alpha$
      - 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
  
  \[
  E \rightarrow T + E_1 \\
  \text{ | } T \\
  T \rightarrow \text{int} \ast T_1 \\
  \text{ | } \text{int}
  \]

  \{ \begin{align*}
  & E.\text{val} = T.\text{val} + E_1.\text{val} \\
  & E.\text{val} = T.\text{val} \\
  & T.\text{val} = \text{int}.\text{val} \ast T_1.\text{val} \\
  & T.\text{val} = \text{int}.\text{val}
  \end{align*} \}

  consider the parsing of the string \ 3 \ast 5 + 8

- Recall: creating the AST

  \[
  E \rightarrow \text{int} \\
  \text{ | } E_1 + E_2 \\
  \text{ | } (E_1)
  \]

  \{ \begin{align*}
  & E.\text{ast} = \text{mkleaf(\text{int}.\text{lexval})} \\
  & E.\text{ast} = \text{mktree(plus, E_1.\text{ast}, E_2.\text{ast})} \\
  & E.\text{ast} = E_1.\text{ast}
  \end{align*} \}

  a bottom-up evaluation of the \text{ast} attribute:

  \[
  E.\text{ast} = \text{mktree(plus, mkleaf(5),} \\
  \text{mktree(plus, mkleaf(2), mkleaf(3))})
  \]
A Hierarchy of Grammars (and Languages)
LALR vs. LR Parsing (LALR < LR)

- LALR(k) is strictly less powerful compared to LR(k)
  - LALR merges and reduces number of states.
- LALR(k) can be slightly less unintuitive compared to LR(k)
  - If there is a reduce-reduce conflict, is it because of merging?
- However, LALR is popular due to its efficiency
  - 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 *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 *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 $C \rightarrow .$
    - Follow($B$) = \{b, c\} and Follow($C$) = \{c\}
  - For LL(1), no conflict
    - First($Bb$) = \{b\}, First($Cc$) = \{c\}, 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. $S \rightarrow a \mid b$ 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 general bottom-up parser for all CFGs that falls back on trial-and-error on conflict
    - L(GLR(k)) == L(CFG)
    - Can be applied to any LR table (e.g. SLR, LALR, Canonical LR)
    - GLR implementations: GNU Bison etc. (but not Yacc)
Notes on Parsing

- Parsing
  - Specifying a language: CFG
  - Automated top-down parser: LL(1)
  - Automated bottom-up parser: LR Family
  - An efficient LR Family parser: LALR(1)
  - LALR(1) parser generators
  - Comparison of subsets of CFG that can be parsed using LL, SLR, LALR, LR, Recursive Descent, GLR
Other Uses of Grammars

Grammars can also be used for

- NLP: parsing natural languages (e.g. English)
- Computational Biology: RNA structure prediction
- Often Probabilistic CFG or Weighted CFG is used

Probabilistic CFG

- Assigns a probability for each rule between 0% -100%
- Good when language allows ambiguity. E.g.
  - In English, “I ate a fish with a fork.” may mean two things:
    1. I ate a fish using a fork. (High probability)
    2. I ate a fish that had a fork inside it. (Low probability)

- RNA may form different structures based on energy levels
- Probabilities can be trained using machine learning techniques