LR Parsing, Part 2
Constructing Parse Tables

Parse table construction
Grammar conflict handling
Categories of LR Grammars and Parsers

Need to Automatically Construct LR Parse Tables: Action and GOTO Table

Construct parse tables from the grammar as follows:
- First build a GOTOgraph (an NFA) to recognize viable prefixes
- Make it deterministic (DFA)
- Then fill in Action and GOTO tables

Example Grammar G
1. \( L \rightarrow L , E \)
2. \( \mid E \)
3. \( E \rightarrow a \)
4. \( \mid b \)

Classes of LR Parsers/Grammars
- LR(0) – Too weak (no lookahead)
- SLR(1) – Simple LR, 1 token lookahead
- LALR(1) – Most common, 1 token lookahead
- LR(1) – 1 token lookahead – big tables
- LR\((k)\) – \(k\) tokens lookahead – even bigger tables

Differences between LR parsers:
- Table size varies widely.
- Errors not discovered as quickly by some variants.
- Different limitations in the language definitions, grammars.

An NFA Recognizing Viable Prefixes

A.k.a. the "characteristic finite automaton" for a grammar G
- States: LR(0) items (= context-free items) of extended Grammar (definition, see next page)
- Input stream: The grammar symbols on the stack
- Start state: \([S' \rightarrow \_ | . S \] \) Final state: \([S' \rightarrow \_ | S ] \)
- Transitions:
  - "move dot across symbol" if symbol found next on stack:
    \( A \rightarrow \alpha . B \gamma \) to \( A \rightarrow \alpha B . \gamma \)
    \( A \rightarrow \alpha . b \gamma \) to \( A \rightarrow \alpha b . \gamma \)
  - \( \epsilon \)-transitions to LR(0)-items for nonterminal productions from items where the dot precedes that nonterminal:
    \( A \rightarrow \alpha . B \gamma \) to \( B \rightarrow \_ \gamma \)

Example Grammar G
1. \( L \rightarrow L , E \)
2. \( \mid E \)
3. \( E \rightarrow a \)
4. \( \mid b \)

Handle, Viable Prefix
- Consider a rightmost derivation \( S \Rightarrow \_ \_ \) \( |Xu = \alpha \_ \_ \) \( |\alpha u \)
  for a context-free grammar G.
- \( \alpha \) is called a handle of the right sentential form \( \beta . u \), associated with the rule \( X \Rightarrow \_ \_ \alpha \)
- Each prefix of \( |\alpha u \) is called a viable prefix of G.

Example: Grammar G with productions \( \{ S \rightarrow aSb \mid c \} \)
- Right sentential forms: e.g. \( c , abc , aSb , aaaaSbbbb , \ldots \)
- For c: Handle: c Viable prefixes: c, c
- For acb: Handle: c Viable prefixes: c, a, ac
- For aSb: Handle: aS Viable prefixes: c, a, aS, aSb
- For aaSbb: Handle: aSb Viable prefixes: c, a, aa, aaS, aaSb
- ...

Right Derivation and Viable Prefixes

Input: a, b, a

Right derivation (handles are underlined, and blue)
\( \langle \text{list} \rangle \Rightarrow \_ \_ \langle \text{list} \rangle , \langle \text{element} \rangle \)
\( \Rightarrow \_ \_ \langle \text{list} \rangle , a \)
\( \Rightarrow \_ \_ \langle \text{list} \rangle , \langle \text{element} \rangle , a \)
\( \Rightarrow \_ \_ \langle \text{list} \rangle , b , a \)
\( \Rightarrow \_ \_ \langle \text{element} \rangle , b , a \)
\( \Rightarrow \_ \_ \langle \text{list} \rangle , b , a \)

Some Viable prefixes of the sentential form: \( \langle \text{list} \rangle , b , a \)
are
\( \{ c , \langle \text{list} \rangle ; , \langle \text{list} \rangle , ; \langle \text{list} \rangle , b ; , \langle \text{list} \rangle , b , ; \langle \text{list} \rangle , b , a \} \)
Definition of LR(0) Item

- An LR(0) item of a rule P is a rule with a dot "•" somewhere in the right side.

Example:
- All LR(0) items of the production
  1. \( <\text{list}> \rightarrow <\text{list}>, <\text{element}> \)
  are
  \( <\text{list}> \rightarrow •, <\text{element}> \), \( <\text{list}> \rightarrow <\text{list}>, •, <\text{element}> \)
  \( <\text{list}> \rightarrow <\text{list}>, •, <\text{element}> \)

Intuitively an item is interpreted as how much of the rule we have found and how much remains.
- Items are put together in sets which become the LR analyser’s state.

Informal Construction of GOTO-Graph

We want to construct a DFA which recognises all viable prefixes of \( G(<\text{sys}>) \):\n
GOTO-graph
(A GOTO-graph is not the same as a GOTO-table but corresponds to an ACTION + GOTO-table. The graph discovers viable prefixes.)

Augmented Grammar \( G(<\text{sys}>) \):
- \( 0. <\text{SYS}> \rightarrow <\text{SYS}>, - \)
- \( 1. <\text{SYS}> \rightarrow <\text{SYS}>, <\text{element}> \)
- \( 2. <\text{element}> \rightarrow •, <\text{element}> \)
- \( 3. <\text{element}> \rightarrow a \)
- \( 4. \mid b \)

Construction Sets of LR(0) Items

Set \( I_0 \): Kernel (Basis)
- \( 0. <\text{SYS}> \rightarrow <\text{SYS}>, - \)
- \( 1. <\text{SYS}> \rightarrow <\text{SYS}>, <\text{element}> \)
- \( 2. <\text{element}> \rightarrow •, <\text{element}> \)
- \( 3. <\text{element}> \rightarrow a \)
- \( 4. \mid b \)

Set \( I_1 \): Additional Closure
- \( 0. <\text{SYS}> \rightarrow <\text{SYS}>, - \)
- \( 1. <\text{SYS}> \rightarrow <\text{SYS}>, <\text{element}> \)
- \( 2. <\text{element}> \rightarrow •, <\text{element}> \)
- \( 3. <\text{element}> \rightarrow a \)
- \( 4. \mid b \)

Set \( I_2 \): Additional Closure
- \( 0. <\text{SYS}> \rightarrow <\text{SYS}>, - \)
- \( 1. <\text{SYS}> \rightarrow <\text{SYS}>, <\text{element}> \)
- \( 2. <\text{element}> \rightarrow •, <\text{element}> \)
- \( 3. <\text{element}> \rightarrow a \)
- \( 4. \mid b \)

Set \( I_3 \): Additional Closure
- \( 0. <\text{SYS}> \rightarrow <\text{SYS}>, - \)
- \( 1. <\text{SYS}> \rightarrow <\text{SYS}>, <\text{element}> \)
- \( 2. <\text{element}> \rightarrow •, <\text{element}> \)
- \( 3. <\text{element}> \rightarrow a \)
- \( 4. \mid b \)

GOTO Graph with States as Sets of LR(0) Items

Based on the canonical collection of LR(0) items draw the GOTO graph.

The GOTO graph discovers those prefixes of right-sentential forms which have (at most) one handle furthest to the right in the prefix.

Example Grammar
- \( 0. L \rightarrow L, E \)
- \( 1. E \rightarrow E \)
- \( 2. E \rightarrow a \)
- \( 3. E \rightarrow b \)
- \( 4. E \rightarrow b \)

Fill in Action Table from GOTO Graph

1. If there is an item \( <\text{A}> \rightarrow \alpha \cdot a \beta \in I_j \) and \( \text{GOTOgraph}(i, x) = I_j \)
2. If there is a complete item (i.e., ends in a dot "•"): \( <\text{A}> \rightarrow \alpha \cdot \beta \in I_j \)
3. If we have \( \text{GOTOgraph}(i, x) \)}
   - accept the symbol \( - \)
4. Otherwise error
Table Differences LR(0), SLR(1), LALR(1)

in which column(s) should reduce x be written?
LR(0) fills in for all input.
SLR(1) fills in for all input in FOLLOW(<A>).
LALR(1) fills in for all those that can follow a certain instance of <A>, see later.

Computing the LR(0) Item Closure (Detailed Algorithm)

For a set I of LR(0) items compute Closure(I) (union of Kernel and Closure):

1. Closure(I) := I (start with the kernel)
2. If [A → α.B] in Closure(I)
   then add [B → γ] to Closure(I) (if not already there)
3. Repeat Step 2 until no more items can be added to Closure(I).

Remarks:
- For s = [A → α.B], Closure(s) contains all NFA states reachable from ε via δ-transitions, i.e., starting from which any substring derivable from B|j| could be recognized. A.k.a. ε-closure(s).
- Then apply the well-known subset construction to transform Closure-NFA -> DFA.
- DFA states will be sets unioning closures of NFA states.

GOTOgraph Function and DFA States Detailed algorithm

Given: Set I of items, grammar symbol X

- \text{GOTOgr}(I, X) := \bigcup [A \rightarrow \alpha.X] \in I \, \text{Closure}(\{[A \rightarrow \alpha.X] \})
  - To become the state transitions in the DFA.
- Now do the subset construction to obtain the DFA states:
  \[ C := \text{Closure}(\{S \rightarrow \cdot |.S| \}) \quad \text{// C: Set of sets of NFA states} \]
  \[ \text{repeat} \]
  - for each set of items I of C:
    - for each grammar symbol X
      - If (GOTOgr(I,X) is not empty and not in C)
        - add GOTOgr(I,X) to C
  \[ \text{until} \] no new states are added to C on a round.

Filling in the GOTO Table

Example Grammar
1. L → L.E
2. L → E
3. E → a
4. E → b

\[ \begin{array}{ccc}
\text{GOTO table:} & \text{Nonterminals} \\
\text{state} & \text{L} & \text{E} \\
0 & 1 & 6 \\
1 & * & 3 \\
2 & * & * \\
3 & * & * \\
4 & * & * \\
5 & * & * \\
\end{array} \]

Representing Sets of Items Implementation in Parser Generator

- Any item [A → α.B] can be represented by 2 integers:
  - production number
  - position of the dot within the RHS of that production
- The resulting sets often contain “closure” items (where the dot is at the beginning of the RHS).
  - Can easily be reconstructed (on demand) from other (“kernel”) items
    - Kernel items: start state [S' \rightarrow \cdot |.S|], plus all items where the dot is not at the left end.
    - Store only kernel items explicitly, to save space.

Resulting DFA

- All states correspond to some viable prefix
- Final states: contain at least one item with dot to the right
  - recognized some handle \rightarrow reduce may (must) follow
- Other states: handle recognition incomplete -> shift will follow
- The DFA is also called the GOTO graph (not the same as the LR GOTO Table!!)
- This automaton is deterministic as a FA (i.e., selecting transitions considering only input symbol consumption) but can still be nondeterministic as a pushdown automaton (e.g., in state I3 above: to reduce or not to reduce?)
From DFA to parser tables:  **ACTION**
**Detailed Algorithm, Summary**

1. For each DFA transition \( I_i \rightarrow I_j \) reading a terminal \( a \) in \( \Sigma \)
   (thus, \( I_i \) contains items of kind \( [X \rightarrow \alpha.a, \beta] \))
   - enter \( S_j \) (shift, new state \( I_j \)) in ACTION[\( I_i, a \)]

2. For each DFA final state \( I_i \)
   (containing a complete item \( [X \rightarrow \alpha.] \))
   - enter \( R_\alpha \) in ACTION table:

<table>
<thead>
<tr>
<th>State</th>
<th>( \alpha )</th>
<th>( \beta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X X S4 B5</td>
<td>S4 S5</td>
</tr>
<tr>
<td>1</td>
<td>A S2 *</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>X X S4 B5</td>
<td>B5</td>
</tr>
<tr>
<td>3</td>
<td>R1 R1 *</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>R3 R3 *</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>R4 R4</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>R2 R2</td>
<td>*</td>
</tr>
</tbody>
</table>

3. For each DFA state containing \( [S' \rightarrow S.|--.] \)
   - enter \( A \) in ACTION[\( I_i, |--. \)] (accept). NB - Conflict? (as in 2.)

**GOTO Table**

<table>
<thead>
<tr>
<th>State</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Conflicts and Categories of LR Grammars and Parsers**

**Conflicts in LR Grammars**

- **Observe conflicts** in DFA (GOTO graph) kernels or at the latest when filling the ACTION table.

  - **Shift-Reduce conflict**
    - A DFA accepting state has an outgoing transition, i.e. contains items \( [X \rightarrow \alpha.] \) and \( [Y \rightarrow \beta.Z] \) for some \( Z \) in \( \text{Nu} \).

  - **Reduce-Reduce conflict**
    - A DFA accepting state can reduce for multiple nonterminals, i.e. contains at least 2 items \( [X \rightarrow \alpha.] \) and \( [Y \rightarrow \beta.] \), \( X \neq Y \).

  - **(Shift/Reduce-Accept conflict)**
    - A DFA accepting state containing \( [S \rightarrow S.|--.] \) contains another item \( [X \rightarrow \alpha.S.|] \) or \( [X \rightarrow \alpha.S.b] \)

Only for LR(0) grammars there are no conflicts.

**Conflict Examples in LR Grammars**

- **Shift – Reduce conflict**:
  - \( E \rightarrow \text{id + E} \) (shift +)
  - \( \text{id} \) (reduce id)

- **Reduce – Reduce conflict**:
  - \( E \rightarrow \text{id} \) (reduce id)
  - \( \text{Pcall} \rightarrow \text{id} \) (reduce id)

- **(Shift – Accept conflict)**
  - \( S' \rightarrow L \) (accept)
  - \( L \rightarrow \text{L} \), \( E \) (shift .)

**Handling Conflicts in LR Grammars**

(Overview):

- Use lookahead
  - if lucky, the LR(0) states + a few fixed lookahead sets are sufficient to eliminate all conflicts in the LR(0)-DFA
    - SLR(1), LALR(1)
  - otherwise, use LR(1) items \( [X \rightarrow \alpha.\beta, a] \) (a is look-ahead) to build new, larger NFA/DFA
    - expensive (many items/states \( \rightarrow \) very large tables)
  - if still conflicts, may try again with \( k > 1 \) \( \rightarrow \) even larger tables

- Rewrite the grammar (factoring / expansion) and retry...

- If nothing helps, re-design your language syntax
  - Some grammars are not LR(\( k \)) for any constant \( k \) and cannot be made LR(\( k \)) by rewriting either

TDDD55 Compilers and Interpreters
TDDD44 Compiler Construction

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.
Look-Ahead (LA) Sets

- For a LR(0) item \([X \rightarrow \alpha \beta]_l\) in DFA-state \(I_l\), define look ahead set \(LA(I_l, [X \rightarrow \alpha \beta]_l)\) (a subset of \(\Sigma\))
- For SLR(1), LALR(1) etc., the LA sets only differ for reduce items.
- For LR(1):
  \[LA_{LR(1)}(I_l, [X \rightarrow \alpha \beta]_l) = \{ a \in \Sigma : \text{S' \Rightarrow^* aXw and the LR(0)-DFA started in } I_l \text{ reaches } I_l \text{ after reading } \mu \} \]
  - usually a subset of \(FOLLOW(X)\), i.e. of SLR LA set
  - depends on state \(I_l\)

Example: L-Values in C Language

- L-values on left hand side of assignment.
  - Part of a C grammar:
    1. \(S \rightarrow S\)
    2. \(S \rightarrow L \rightarrow R\)
    3. \(L \rightarrow R\)
    4. \(L \rightarrow \ast R\)
    5. \(\ast id\)
    6. \(R \rightarrow L\)

- Avoids that \(R\) (for R-values) appears as LHS of assignments
- But \(\ast R \ldots\) is ok.

- This grammar is LALR(1) but not SLR(1):

Made it simple: Is my grammar SLR(1)?

- Construct the (LR(0)-item) characteristic NFA and its equivalent DFA (= GOTO graph) as above.
- Consider all conflicts in the DFA states:
  - Shift-Reduce:
    \[\frac{\text{Shift-Reduce:}}{\text{Consider all pairs of conflicting items } [X \rightarrow \alpha \beta], [Y \rightarrow \beta \gamma]; \text{If } b \in FOLLOW(X) \text{ for any of these } \rightarrow \text{not SLR(1).}}\]
  - Reduce-Reduce:
    \[\frac{\text{Reduce-Reduce:}}{\text{Consider all pairs of conflicting items } [X \rightarrow \alpha \beta], [Y \rightarrow \beta \gamma]; \text{If FOLLOW}(X) \text{ intersects with FOLLOW}(Y) \rightarrow \text{not SLR(1)}}\]
  - (Shift-Accept: similar to Shift-Reduce)

Example (cont.)

- LR(0) parser has a shift-reduce conflict in kernel of state \(I_2\):
  \[I_2 = \{ [S \rightarrow S], [S \rightarrow \ast R], [S \rightarrow R], [L \rightarrow \ast R], [L \rightarrow \ast \ast R], [L \rightarrow \ast id], [L \rightarrow \ast \ast id], [R \rightarrow \ast L], [R \rightarrow \ast \ast L], [R \rightarrow \ast \ast \ast L] \}
  \[I_3 = \{ [S \rightarrow S], [S \rightarrow \ast R], [S \rightarrow R], [L \rightarrow \ast R], [L \rightarrow \ast \ast R], [L \rightarrow \ast id], [L \rightarrow \ast \ast id], [R \rightarrow \ast L], [R \rightarrow \ast \ast L], [R \rightarrow \ast \ast \ast L] \}
  \[I_4 = \{ [L \rightarrow \ast R], [L \rightarrow \ast \ast R], [L \rightarrow \ast id], [L \rightarrow \ast \ast id], [R \rightarrow \ast L], [R \rightarrow \ast \ast L], [R \rightarrow \ast \ast \ast L] \}
  \[I_5 = \{ [S \rightarrow \ast R], [S \rightarrow \ast \ast R], [S \rightarrow \ast id], [S \rightarrow \ast \ast id], [L \rightarrow \ast R], [L \rightarrow \ast \ast R], [L \rightarrow \ast id], [L \rightarrow \ast \ast id], [R \rightarrow \ast L], [R \rightarrow \ast \ast L], [R \rightarrow \ast \ast \ast L] \}
  \[I_6 = \{ [L \rightarrow \ast R], [L \rightarrow \ast \ast R], [L \rightarrow \ast id], [L \rightarrow \ast \ast id], [R \rightarrow \ast L], [R \rightarrow \ast \ast L], [R \rightarrow \ast \ast \ast L] \}
  \[I_7 = \{ [R \rightarrow \ast L], [R \rightarrow \ast \ast L], [R \rightarrow \ast \ast \ast L] \}
  \[I_8 = \{ [L \rightarrow \ast \ast id], [L \rightarrow \ast \ast \ast id], [R \rightarrow \ast \ast \ast L] \}
  \[I_9 = \{ [L \rightarrow \ast \ast \ast id], [R \rightarrow \ast \ast \ast \ast L] \}
  \[LA_{LR(1)}(I_2, [R \rightarrow \ast L]) = \{ [\rightarrow] \} \rightarrow \text{SLR(1)} \text{ still shift-reduce conflict in } I_2 \text{ as } \rightarrow \text{does not disambiguate}

Example (cont.)

- LALR(1) Parser Construction
  - Method 1: (simple but not practical)
    1. Construct the LR(1) items (see later). (If there is already a conflict, stop.)
    2. Look for sets of LR(1) items that have the same kernel, and merge them.
    3. Construct the ACTION table as for LR(1).
       If a conflict is detected, the grammar is not LALR(1).
    4. Construct the GOTO graph function:
       For each merged \(J = I_1 \cup I_2 \cup ... \cup I_n\)
       - the kernels of GOTOGr(\(i_j, X\)) = identical because the kernels of \(I_1, ..., I_n\) are identical.
       - Set GOTOGr(\(J, X\)) = \(U \{ I_j \text{ has the same kernel as GOTOGr}(I_j, X) \})
  - Method 2: (practical, used) (details see textbook)
    1. Start from LR(0) items and construct kernels of DFA states \(I_0, I_1, ...
    2. Compute lookahead sets by propagation along the GOTOGr(\(i_j, X\)) edges (fixed point iteration).
**Solve Conflicts by Rewriting the Grammar**

- **Eliminate Reduce-Reduce Conflict:**
  - **Factoring**
    
    $S \rightarrow (A) \mid (B)$
    
    $A \rightarrow \text{char} \mid \text{integer} \mid \text{ident}$
    
    $B \rightarrow \text{float} \mid \text{double} \mid \text{ident}$
  
  - **Eliminate Shift-Reduce Conflict:** (one token lookahead: ‘’)
    
    **Inline-Expansion**
    
    $S \rightarrow \{ A \} \mid \text{OptY} \{ B \}$
    
    $\text{OptY} \rightarrow Y \{ \}$
    
    $Y \rightarrow \ldots$
    
    $A \rightarrow \ldots$
    
    $B \rightarrow \ldots$

**Some grammars are not LR(k) for any fixed k**

- Example: $S \rightarrow a \ B \ c$
  
  $B \rightarrow b \ B \ b \ b \ | \ b$
  
  describes language $\{a^b b^r c^s : s \geq 0\}$

  - This grammar is not LR(k) for any fixed k.

  **Proof:** As k is fixed (constant), consider for any $n > k$:
    
    - $S \Rightarrow^* a^b b^r c^s = a^b (b^r)^n c^s$
    - $S \Rightarrow^* a^b b^r (b^r)^n c = a^b b^{r+1} b^{r^n} c$

  By the LR(k) definition:
    
    - $\alpha = a^b$
    - $\beta = b$
    - $\gamma = b^{r+1} b^{r^n} c$

  Although $w[1:k] = y[1:k]$ and production rule $S' \Rightarrow^* Y = Y$ imply $\alpha = \gamma$ and $x = y = w$. The handle cannot be localized with only limited lookahead size k.

- Although $w[1:k] = y[1:k]$, we have $\alpha \Rightarrow^* \gamma$; grammar is not LR(k).

**LR(k) Grammar - Formal Definition**

- Let $G'$ be the augmented grammar for G (i.e., extended by new start symbol $S'$ and production rule $S' \Rightarrow S$)

  - G is called a LR(k) grammar if
    
    - $S' \Rightarrow^* a \ X \ W \Rightarrow^* a \ j w$ and
    - $S' \Rightarrow^* \gamma X \ W \Rightarrow^* a \ j \ y$ and
    - $w[1:k] = y[1:k]$

  imply that $\alpha \Rightarrow^* \gamma$ and $X = Y$ and $x = y = w$. Remark: $w, x, y \in \Sigma^*$

**No ambiguous grammar is LR(k) for any fixed k**

- **S** \rightarrow \text{if} E \text{ then } S \text{ else } S \text{ other statements...}

  is ambiguous – the following statement has 2 parse trees:
    
    - if $E_1$ then if $E_2$ then $S_1$ else $S_2$

**Rewriting the grammar**...
Some grammars are not LR(k) for any fixed k

- Grammar with productions
  \[ S \rightarrow a S a | \varepsilon \]
  is unambiguous but not LR(k) for any fixed k. (Why?)

- An equivalent LR grammar for the same language is
  \[ S \rightarrow a a S | \varepsilon \]

LR(1) Items and LR(k) Items

LR(k) parser: Construction similar to LR(0) / SLR(1) parser, but plan for distinguishing between states for \( k \geq 0 \) tokens lookahead already from the beginning

- States in the LR(0) GOTO graph may be split up

LR(1) items:
- \([ A \rightarrow \alpha \beta, a ]\) for all productions \( A \rightarrow \alpha \beta \) and all \( a \in \Sigma \)
- Can be combined for lookahead symbols with equal behavior:
  \([ A \rightarrow \alpha \beta, a[b] ]\) or \([ A \rightarrow \alpha \beta, L ]\) for a subset \( L \) of \( \Sigma \)
- Generalized to \( k > 1 \):
  \([ A \rightarrow \alpha \beta, a_1a_2...a_k ]\)

Interpretation of \([ A \rightarrow \alpha \beta, a ]\) in a state:
- If \( \beta \) not \( \varepsilon \), ignore second component (as in LR(0))
- If \( \beta = \varepsilon \), i.e., \([ A \rightarrow a_1...a_k ]\), reduce only if next input symbol = \( a \)

LR(1) Parser

- NFA start state is \([ S' \rightarrow S, \cdot ]\)
- Modify computation of \( \text{Closure}(I) \), \( \text{GOTO}(I,X) \) and the subset computation for LR(1) items
  - Details see [ASU86, p.232] or [ALSU06, p.261]
- Can have many more states than LR(0) parser
  - Which may help to resolve some conflicts

Interesting to know...

- For each LR(k) grammar with some constant \( k \geq 1 \) there exists an equivalent* grammar \( G' \) that is LR(1).
- For any LL(k) grammar there exists an equivalent LR(k) grammar (but not vice versa!)
  - e.g., language \( \{ a^n b^n : n > 0 \} \cup \{ a^n c^n : n > 0 \} \)
    has a LR(0) grammar but no LL(k) grammar for any constant \( k \).
- Some grammars are LR(0) but not LL(k) for any \( k \)
  - e.g., \( S \rightarrow A b \)
    \( A \rightarrow a A | a \) (left recursion, could be rewritten)

* Two grammars are equivalent if they describe the same language.