LR(1) Shift-Reduce Parsing

Roadmap (Where are we?)

Last lecture
- Bottom-up parsing
  - Finding reductions
  - Shift-reduce parsers

This lecture
- Shift-reduce parser
  - Parsing with ACTION/GOTO tables
- LR(1) parsing
  - LR(1) items
  - Computing closure
  - Computing goto
  - LR(1) canonical collection

---

LR(1) Skeleton Parser

```java
stack.push(INVALID); stack.push(s0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
  s = stack.top();
  if (ACTION[s,token] == "reduce A →β") then {
    stack.popnum(2*|β|);
    s = stack.top();
    stack.push(A);
    stack.push(GOTO[s,A]);
  } else if (ACTION[s,token] == "shift s1") then {
    stack.push(token); stack.push(s1);
    token ← scanner.next_token();
  } else if (ACTION[s,token] == "accept" & token == $) then not_found = false;
  else report a syntax error and recover;
}
report success;
```

The skeleton parser
- uses ACTION & GOTO tables
- does (world) shifts
- does (derivation) reductions
- does I accept
- detects errors by failure of 3 other cases

---

Extended / Augmented Grammar

Algorithm
- If start symbol S of grammar appears in the right-hand side of a production
  - Introduce new start symbol S' and the production S' → S
The result is an extended or augmented grammar
Example
- Original grammar
  - 1. SheepNoise → SheepNoise baa
  - 2. SheepNoise → SheepNoise baa
  - 3. SheepNoise → baa
- Augmented grammar
  - 1. SheepNoise → SheepNoise baa
  - 2. SheepNoise → SheepNoise baa
  - 3. SheepNoise → baa

---

LR(1) Parsers (parse tables)

To make a parser for L(G), need a set of tables

The grammar

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SheepNoise → SheepNoise baa</td>
<td>SheepNoise baa</td>
<td>SheepNoise baa</td>
</tr>
<tr>
<td>2</td>
<td>SheepNoise → SheepNoise baa</td>
<td>SheepNoise baa</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SheepNoise → SheepNoise baa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>$</th>
<th>ba</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift 2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>accept</td>
<td>shift 3</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3</td>
<td>reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>reduce 2</td>
<td>reduce 2</td>
</tr>
</tbody>
</table>

#s for shift are state #s
#s for reduce are grammar production #s

---

Example Parse 1

The string "baa"

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SheepNoise → SheepNoise baa</td>
<td>SheepNoise baa</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SheepNoise → SheepNoise baa</td>
<td>SheepNoise baa</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SheepNoise → baa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ACTION

<table>
<thead>
<tr>
<th>State</th>
<th>$</th>
<th>ba</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift 2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>accept</td>
<td>shift 3</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3</td>
<td>reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>reduce 2</td>
<td>reduce 2</td>
</tr>
</tbody>
</table>

GOTO

<table>
<thead>
<tr>
<th>State</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Stack Input Action

| $s_0 | shift 1 | shift 2 |
| $s_1 | reduce 3 | reduce 3 |
| $s_2 | accept |    |

---

Remember, this is the left-recursive SheepNoise. EaC shows the right-recursive version.
LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include top of stack (TOS)
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - To match subterm, invoke subterm
    - Build up the states and transition functions of the DFA
- Build a handle-recognizing DFA
  - ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA
  - & leave old DFA’s state on stack
- Final state in DFA ⇒ a reduce action
  - New state is GOTO( ) or TOS (other-pg. No)
  - For SN this takes the DFA to A

LR(1) items

The LR(1) table construction algorithm uses LR(1) items to represent valid configurations of an LR(1) parser

An LR(1) item is a pair [P, B], where
- P is a production A → α with α at some position in the rhs
- B is a lookahead string of length ≤ k

The • in an item indicates the position of the top of the stack

LR(1)

- [A → α][ γ] means that the input seen so far is consistent with the use of A → α immediately after the symbol on top of the stack
- (A → α)[ γ] means that the input seen so far is consistent with the use of A → α at this point in the parse, and that the parser has already recognized β.
- [A → α γ] means that the parser has seen γ, and that a lookahead symbol of γ is consistent with reducing to A.

Building LR(1) Parsers

How do we generate the ACTION and GOTO tables?

- Use the grammar to build a model of the DFA
- Use the model to build ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)

The Big Picture

- Model the state of the parser
- Use two functions goto(s, X) and closure(s)
  - goto(s) is analogous to move( ) in the subset construction
  - closure(s) adds information to round out a state
- Build up the states and transition functions of the DFA
- Use this information to fill in the ACTION and GOTO tables

LR(1) Items

The production A → β, where β = B B, with lookahead γ, can give rise to 4 items

- [A → B][ γ]
- [A → B][ γ]
- [A → B][ γ]
- [A → B][ γ]

The set of LR(1) items for a grammar is finite

What’s the point of all these lookahead symbols?

- Carry them along to choose the correct reduction, if there is a choice
- Lookaheads are bookkeeping, unless item has • at right end
  - Has no direct use in [A → γ]
  - In [A → γ], a lookahead of γ implies a reduction by A → B
- For ([A → γ][β → γ β]), γ → reduce to A, FIRST(γ) = shift
  - Limited right context is enough to pick the actions (unique, i.e., deterministic choice)
LR(1) Table Construction

High-level overview
1. Build the canonical collection of sets of LR(1) Items, \( I \)
   - Begin in an appropriate state, \( s_0 \)
     - Assume: \( S \rightarrow \delta S' \) and \( S' \) is unique start symbol that does not occur on any RHS of a production (extended CFG - ECFG)
     - \( [S' \rightarrow \delta S] \) along with any equivalent items
     - Derive equivalent items as \( \text{closure}(s_0) \)
   - Repeatedly compute, for each \( s_k \) and each \( X, \text{goto}(s_k, X) \)
     - If the set is not already in the collection, add it
     - Record all the transitions created by \( \text{goto}(s_0) \)
   - This eventually reaches a fixed point
2. Fill in the table from the collection of sets of LR(1) items
   - The canonical collection completely encodes the transition diagram for the handle-finding DFA

Back to Finding Handles

Revisiting an issue from last class
Parser in a state where the stack (the fringe) was
\( \text{Expr} \rightarrow \text{Term} \)
With lookahead of \( \ast \)
How did it choose to expand \( \text{Term} \) rather than reduce to \( \text{Expr} \)?
• Lookahead symbol is the key
  • With lookahead of \( + \) or \( - \), parser should reduce to \( \text{Expr} \)
  • With lookahead of \( \ast \) or \( / \), parser should shift
• Parser uses lookahead to decide
• All this context from the grammar is encoded in the handle recognizing mechanism

Computing Closures

\( \text{Closure}(s) \) adds all the items implied by items already in \( s \)
- Any item \( [A \rightarrow \delta B \delta a] \) implies \( [B \rightarrow \tau, x] \) for each production with \( B \) on the lhs, and each \( x \in \text{FIRST}(\delta a) \)
The algorithm
\[
\text{Closure}(s) \quad \text{while (} s \text{ is still changing)}
\]
- \( \forall \) items \( [A \rightarrow \delta B \delta a] \in s \) // item with \( \ast \) to left of NT \( B \)
  - \( \forall \) productions \( B \rightarrow \tau \in P \) // all productions for \( B \)
  - \( \forall b \in \text{FIRST}(\delta a) \) // tokens appearing after \( B \)
  - if \( [B \rightarrow \tau, b] \notin s \) // form LR(1) item w/ new lookahead
    - then add \( [B \rightarrow \tau, b] \) to \( s \) // add item to \( s \) if new

Example - Closures With LR(0) Items

Grammar
- \( P_0 \)
- \( P_1 \)
- \( P_2 \)
- \( P_3 \)

Sets of LR(0) items
- \( S \rightarrow \ast E \)
- \( E \rightarrow T \ast E \)
- \( T \rightarrow \ast id \)
- \( \text{FIRST}(\ast) = \{ \ast \} \)

Example - Closures With LR(1) Items

Grammar
- \( P_0 \)
- \( P_1 \)
- \( P_2 \)
- \( P_3 \)

Sets of LR(1) items
- \( S \rightarrow \ast E, \ast \)
- \( E \rightarrow T \ast E, \ast, \)
- \( T \rightarrow \ast id, \ast \)
- \( \text{FIRST}(\ast) = \{ \ast \} \)
- \( \text{FIRST}(\ast) = \{ \ast \} \)
Computing Gotos

\[ \text{Goto}(s, x) \text{ computes the state that the parser would reach if it recognized an } x \text{ while in state } s \]

- \[ \text{Goto}([A \rightarrow \beta \cdot X \delta, a], X) \text{ produces } [A \rightarrow \beta \cdot X \delta, a] \] (easy part)
- Should also include \( \text{closure}([A \rightarrow \beta \cdot X \delta, a]) \) (fill out the state)

The algorithm

\[
\text{Goto}(s, X) \\
\text{new} \leftarrow \emptyset \\
\forall \text{ items } [A \rightarrow \beta \cdot X \delta, a] \in s \hspace{1cm} \text{// for each item with } \cdot \text{ to left of } X \\
\text{new} \leftarrow \text{new} \cup [A \rightarrow \beta \cdot X \delta, a] \hspace{1cm} \text{// add item with } \cdot \text{ to right of } X \\
\text{return closure(new)} \hspace{1cm} \text{// remember to compute closure!}
\]

Example - Goto With LR(0) Items

Grammar

\[
P_0: S' ::= E \\
P_1: E ::= T \cdot E \\
P_2: T ::= \text{id} \\
P_3: T ::= T \cdot E \\
\]

Sets of LR(0) items

\[
\begin{align*}
S' \rightarrow E, A &= \text{id} \cdot E, A \\
E \rightarrow \text{id}, E, \text{id} \cdot E, A \rightarrow \cdot E, E, \text{id} \\
T \rightarrow \text{id}, \text{id} \cdot E \\
\end{align*}
\]

Example - Goto With LR(1) Items

Grammar

\[
P_0: S' ::= E \\
P_1: E ::= T \cdot E \\
P_2: T ::= \text{id} \\
P_3: T ::= T \cdot E \\
\]

Sets of LR(1) items

\[
\begin{align*}
S' \rightarrow E, A &= \text{id} \cdot E, A \\
E \rightarrow \text{id}, E, \text{id} \cdot E, A \rightarrow \cdot E, E, \text{id} \\
T \rightarrow \text{id}, \text{id} \cdot E \\
\end{align*}
\]

Building the Canonical Collection

Start from \( s_0 = \text{closure}([S' \rightarrow S, A]) \)

Repeatedly construct new states, until all are found

The algorithm

\[
\text{cc}_0 \leftarrow \text{closure}([S' \rightarrow S, A]) \\
\text{CC} \leftarrow \{ \text{cc}_0 \} \\
\text{while } (\text{new sets are still being added to CC}) \\
\text{for each unmarked set } \text{cc}_j \in \text{CC} \\
\text{mark } \text{cc}_j \text{ as processed} \\
\text{for each } x \text{ following a } \cdot \text{ in an item in } \text{cc}_j \\
\text{temp} \leftarrow \text{goto}(\text{cc}_j, x) \\
\text{if temp } \notin \text{ CC} \\
\text{then } \text{CC} \leftarrow \text{CC} \cup \{ \text{temp} \} \\
\text{record transitions from } \text{cc}_j \text{ to temp on } x
\]

Example - Canonical Collection of LR(0) Items

Example - Canonical Collection of LR(1) Items
The SheepNoise Grammar (revisited)

We will use this grammar again:

\[
\text{Goal} \rightarrow \text{SheepNoise} \\
\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \\
\text{SheepNoise} \rightarrow \ baa \\
\text{SheepNoise} \\
\]

Example From SheepNoise

Initial step builds the item \([\text{Goal} \rightarrow \text{SheepNoise}, \$]\) and takes its closure:

\[
\text{Closure}\left([\text{Goal} \rightarrow \text{SheepNoise}, \$]\right)
\]

So, \(c_0\) is \([\text{Goal} \rightarrow \text{SheepNoise}, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \ baa]\).

Goto\(c_0, \text{baa}\)

- Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{baa} \cdot, $])</td>
<td>Item 3 in (c_0)</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{baa} \cdot, \text{baa}])</td>
<td>Item 5 in (c_0)</td>
</tr>
</tbody>
</table>

- Closure adds nothing since \(\cdot\) is at end of rhs in each item

In the construction, this produces \(c_2\):

\([\text{SheepNoise} \rightarrow \text{baa} \cdot, \$], [\text{SheepNoise} \rightarrow \text{baa} \cdot, \text{baa}]\)

Example From SheepNoise

\[
\text{Goal} \rightarrow \text{SheepNoise} \\
\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \\
\text{SheepNoise} \rightarrow \ baa \\
\text{SheepNoise} \\
\]

Example from SheepNoise

\(c_0\) is \([\text{Goal} \rightarrow \text{SheepNoise}, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \text{baa}]\).

Goto\(c_0, \text{baa}\)

- Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{baa} \cdot, $])</td>
<td>Item 3 in (c_0)</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{baa} \cdot, \text{baa}])</td>
<td>Item 5 in (c_0)</td>
</tr>
</tbody>
</table>

- Closure adds nothing since \(\cdot\) is at end of rhs in each item

In the construction, this produces \(c_2\):

\([\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \cdot, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \cdot, \text{baa}]\)

Example from SheepNoise

\[
\text{Goal} \rightarrow \text{SheepNoise} \\
\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \\
\text{SheepNoise} \rightarrow \ baa \\
\text{SheepNoise} \\
\]

Example from SheepNoise

\(c_0\) is \([\text{Goal} \rightarrow \text{SheepNoise}, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \text{baa}]\).

Goto\(c_0, \text{baa}\)

- Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{baa} \cdot, $])</td>
<td>Item 3 in (c_0)</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \text{baa} \cdot, \text{baa}])</td>
<td>Item 5 in (c_0)</td>
</tr>
</tbody>
</table>

- Closure adds nothing since \(\cdot\) is at end of rhs in each item

In the construction, this produces \(c_2\):

\([\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \cdot, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \cdot, \text{baa}]\)

Starts with \(c_0\)

\(c_1 = \text{Goto}(c_0, \text{SheepNoise}) = \{ [\text{Goal} \rightarrow \text{SheepNoise}, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \$] \}

\(c_2 = \text{Goto}(c_1, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{baa} \cdot, \$], [\text{SheepNoise} \rightarrow \ baa \cdot, \$] \}

Iteration 1 computes

\(c_2 = \text{Goto}(c_2, \text{SheepNoise}) = \{ [\text{Goal} \rightarrow \text{SheepNoise}, \$], [\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa, \$], [\text{SheepNoise} \rightarrow \ baa, \$] \}

\(c_3 = \text{Goto}(c_2, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{baa} \cdot, \$], [\text{SheepNoise} \rightarrow \ baa \cdot, \$] \}

Iteration 2 computes

\(c_3 = \text{Goto}(c_3, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{baa} \cdot, \$] \}

Nothing more to compute, since \(\cdot\) is at the end of every item in \(c_3\).