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

```c
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|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
    }
    else if (ACTION[s,token] == "shift s_i") then {
        stack.push(token); stack.push(s_i);
        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 |words| shifts
- does |derivation| reductions
- does 1 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. baa
  - Augmented grammar
    1. Goal → SheepNoise
    2. SheepNoise → SheepNoise baa
    3. 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>Goal → SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SheepNoise → SheepNoise baa</td>
</tr>
<tr>
<td>3</td>
<td>baa</td>
</tr>
</tbody>
</table>

The tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>SheepNoise</td>
</tr>
<tr>
<td>0</td>
<td>shift 2</td>
</tr>
<tr>
<td>1</td>
<td>accept shift 3</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3 reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>reduce 2 reduce 2</td>
</tr>
</tbody>
</table>

Example Parse 1

The string "baa"

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>SheepNoise</td>
</tr>
<tr>
<td>0</td>
<td>shift 2</td>
</tr>
<tr>
<td>1</td>
<td>accept shift 3</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3 reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>reduce 2 reduce 2</td>
</tr>
</tbody>
</table>

Stack | Input | Action |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>baa $</td>
<td>$</td>
</tr>
<tr>
<td>$s_0$ baa $s_2$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$s_0$ baa $s_1$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>
Example Parse 2

The string "baa baa"

<table>
<thead>
<tr>
<th>State</th>
<th>$</th>
<th>baa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>shift 2</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>

---

Example Parse 3

The string "id + id"

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Stack | Input | Action |
---|------|------|
$ s_0$ | baa baa EOF | shift 2 |
$ s_0$ baa $ s_2$ | baa EOF | reduce 3 |
$ s_0$ baa $ s_1$ | baa EOF | shift 3 |
$ s_0$ baa $ s_1$ | EOF | reduce 2 |
$ s_2$ baa $ s_1$ | EOF | accept |
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)
  → 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[state at TOS (after pop), \( \langle \alpha \rangle \)]
  → For \( SN \), this takes the DFA to \( s \)

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 \( \text{goto}(s, \chi) \) and \( \text{closure}(s) \)
  → \( \text{goto}() \) is analogous to move() in the subset construction
  → \( \text{closure}() \) 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 $A \to \beta$, where $\beta = B_1 B_2 B_3$ with lookahead $a$, can give rise to 4 items:

- $[A \to B_1 B_2 B_3, a]$
- $[A \to B_1 B_2 B_3 a]$
- $[A \to B_1 B_3 B_2 a]$
- $[A \to B_1 B_2 B_3 a]$

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 $\cdot$ at right end:
  - Has no direct use in $[A \to \beta \gamma a]$
  - In $[A \to \beta \cdot a]$, a lookahead of $a$ implies a reduction by $A \to \beta$
  - For $\{ [A \to \beta \cdot a], [B \to \gamma \delta b] \}$, $a \Rightarrow \text{reduce to } A; \text{FIRST}(\delta) \Rightarrow \text{shift}$

⇒ 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, \( I \)
   a. Begin in an appropriate state, \( s_0 \)
      - Assume: \( S' \rightarrow S \) and \( S' \) is unique start symbol that does not occur on any RHS of a production (extended CFG - ECFG)
      - \( [S' \rightarrow S, \epsilon] \), along with any equivalent items
      - Derive equivalent items as \( \text{closure}(s_0) \)
   b. 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 \( \text{goto}() \)
      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 \( * \)

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 \( * \) or \( / \), parser should shift
- Parser uses lookahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism
Lecture 4 15 CS430

Back to $x - 2 * y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id$</td>
<td>- num * id</td>
<td>9,1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Factor$</td>
<td>- num * id</td>
<td>7,1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Term$</td>
<td>- num * id</td>
<td>4,1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$Expr$</td>
<td>- num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr - num$</td>
<td>num * id</td>
<td>8,3</td>
<td>red. 8</td>
</tr>
<tr>
<td>$Expr - Factor$</td>
<td>+ * id</td>
<td>7,3</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Expr - Term$</td>
<td>+ * id</td>
<td>10</td>
<td>shift *</td>
</tr>
<tr>
<td>$Expr - Term$</td>
<td>- * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr - Term$</td>
<td>- * id</td>
<td>9,5</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Expr - Term$</td>
<td>+ * id</td>
<td>5,5</td>
<td>red. 5</td>
</tr>
<tr>
<td>$Expr - Term$</td>
<td>+ * id</td>
<td>3,3</td>
<td>red. 3</td>
</tr>
<tr>
<td>$Expr$</td>
<td>1,1</td>
<td>red. 1</td>
<td></td>
</tr>
<tr>
<td>$Goal$</td>
<td>none</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until TOS is the right end of a handle
2. Find the left end of the handle & reduce

Computing Closures

Closure(s) adds all the items implied by items already in s

- Any item $[A \rightarrow \beta \cdot B \cdot \delta, a]$ implies $[B \rightarrow \tau, x]$ for each production with $B$ on the lhs, and each $x \in \text{FIRST}(\delta a)$

The algorithm

```plaintext
Closure(s) while (s is still changing)
∀ items $[A \rightarrow \beta \cdot B \cdot \delta, a] \in s$ // item with * to left of NT B
∀ productions $B \rightarrow \tau \in P$ // all productions for $B$
∀ $b \in \text{FIRST}(\delta a)$ // tokens appearing after $B$
if $[B \rightarrow \tau, b] \in s$ // form LR(1) item w/ new lookahead
    then add $[B \rightarrow \tau, b]$ to s // add item to s if new
```

- Classic fixed-point method
- Halts because $s \subset \text{ITEMS}$ (worklist version is faster)

Closure "fills out" a state
Example - Closures With LR(0) Items

Grammar

| P0 | S' ::= E |
| P1 | E ::= T + E |
| P2 | T |
| P3 | T ::= id |

Sets of LR(0) items

[S' ::= • E],
[E ::= • T + E],
[T ::= • T],
[T ::= • id]

Example - Closures With LR(1) Items

Grammar

| P0 | S' ::= E |
| P1 | E ::= T + E |
| P2 | T |
| P3 | T ::= id |

Sets of LR(1) items

[S' ::= • E, $],
[E ::= • T + E, $],
[E ::= • T, $],
[T ::= • id, +]

First(ε $) = $
First(+ $) = +

First(ε $) = $
First(+ $) = +
Computing Gotos

$Goto(s,x)$ computes the state that the parser would reach if it recognized an $x$ while in state $s$.

- $Goto([A\to\beta\cdot X\delta.a],X)$ produces $[A\to\beta.X\delta.a]$ (easy part)
- Should also includes $closure([A\to\beta.X\delta.a])$ (fill out the state)

The algorithm

$$\begin{align*}
\text{Goto}(s, X) & \\
\text{new} & \leftarrow \emptyset \\
\forall \text{ items } [A\to\beta\cdot X\delta.a] \in s & \quad \text{// for each item with } \cdot \text{ to left of } X \\
\text{new} & \leftarrow \text{new} \cup [A\to\beta.X\delta.a] \quad \text{// add item with } \cdot \text{ to right of } X \\
\text{return closure(new)} & \quad \text{// remember to compute closure!}
\end{align*}$$

- Not a fixed-point method!
- Straightforward computation
- Uses closure()
  - Goto() moves forward

Example - Goto With LR(0) Items

**Grammar**

| P0 | S' ::= E |
| P1 | E ::= T + E |
| P2 | E ::= T |
| P3 | T ::= id |

**Sets of LR(0) items**

- $[S' ::= E •]$
- $[E ::= • T + E •]$
- $[E ::= • T •]$
- $[T ::= • id •]$

- $[S' ::= E •]$
- $[E ::= • T + E •]$
- $[E ::= • T •]$
- $[T ::= • id •]$
Example – Goto With LR(1) Items

Grammar

| P0 | S' ::= E |
| P1 | E ::= T + E |
| P2 | T ::= id |
| P3 | T ::= id + |

Sets of LR(1) items

- [S' ::= E , $]
- [E ::= • T + E , $]
- [E ::= • T , $]
- [T ::= • id + ]
- [T ::= • id , $]
- [T ::= id , + ]
- [T ::= id , $]

Building the Canonical Collection

Start from $s_0 = closure([S'\rightarrow S , \$])$
Repeatedly construct new states, until all are found

The algorithm

```
cc_0 \leftarrow closure([S'\rightarrow S , \$])
CC \leftarrow \{ cc_0 \}
while (new sets are still being added to CC)
    for each unmarked set cc_j \in CC
        mark cc_j as processed
        for each x following a • in an item in cc_j
            temp \leftarrow goto(cc_j , x)
            if temp \in CC
                then CC \leftarrow CC \cup \{ temp \}
            record transitions from cc_j to temp on x
```

Fixed-point computation
(worklist version)
- Loop adds to CC
- CC \subseteq 2^{ITEMS}, so CC is finite
Example – Canonical Collection of LR(0) Items

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

We will use this grammar again:

\[
\begin{align*}
\text{Goal} & \rightarrow \text{SheepNoise} \\
\text{SheepNoise} & \rightarrow \text{SheepNoise baa} \\
& \quad | \quad \text{baa}
\end{align*}
\]

Example From SheepNoise

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

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

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Goal} \rightarrow \cdot \text{SheepNoise}, $])</td>
<td>Original item</td>
</tr>
</tbody>
</table>
| \([\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa}, \$]\) | 1, \&a is \$
| \([\text{SheepNoise} \rightarrow \cdot \text{baa}, \$]\)       | 1, \&a is \$
| \([\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa}, \text{baa}]\) | 2, \&a is baa $
| \([\text{SheepNoise} \rightarrow \cdot \text{baa}, \text{baa}]\) | 2, \&a is baa $

So, \(c_{c_0}\) is

\[
\{ \text{Goal} \rightarrow \cdot \text{SheepNoise}, \$, \text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa}, \$, \text{SheepNoise} \rightarrow \cdot \text{baa}, \$, \text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa}, \$, \text{SheepNoise} \rightarrow \cdot \text{baa}, \$\}
\]
Example from SheepNoise

$cc_0$ is \{ [Goal $\rightarrow$ SheepNoise $\cdot$, $\$$], [SheepNoise $\rightarrow$ SheepNoise $\cdot$ baa, $\$$] \}
\[SheepNoise $\rightarrow$ baa, $\$$], [SheepNoise $\rightarrow$ SheepNoise baa, baa] \}
\[SheepNoise $\rightarrow$ baa, baa] \}

$Goto( cc_0, baa )$

- Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SheepNoise $\rightarrow$ baa, $$$]</td>
<td>Item 3 in $cc_0$</td>
</tr>
<tr>
<td>[SheepNoise $\rightarrow$ baa, baa]</td>
<td>Item 5 in $cc_0$</td>
</tr>
</tbody>
</table>

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

In the construction, this produces $cc_2$
\{ [SheepNoise $\rightarrow$ baa $\cdot$, ($\$$ baa)] \}

Example from SheepNoise

Starts with $cc_0$
$cc_0$: \{ [Goal $\rightarrow$ SheepNoise $\cdot$, $\$$], [SheepNoise $\rightarrow$ SheepNoise baa, $\$$] \}
\[SheepNoise $\rightarrow$ baa, $\$$], [SheepNoise $\rightarrow$ SheepNoise baa, baa] \}
\[SheepNoise $\rightarrow$ baa, baa] \}

Iteration 1 computes
$cc_1 = Goto(cc_0, SheepNoise) =$
\{ [Goal $\rightarrow$ SheepNoise $\cdot$, $\$$], [SheepNoise $\rightarrow$ SheepNoise $\cdot$ baa, $\$$] \}
\[SheepNoise $\rightarrow$ SheepNoise $\cdot$ baa, baa] \}

$cc_2 = Goto(cc_0, baa) =$ \{ [SheepNoise $\rightarrow$ baa $\cdot$, $\$$], [SheepNoise $\rightarrow$ baa $\cdot$, baa] \}

Iteration 2 computes
$cc_3 = Goto(cc_1, baa) =$ \{ [SheepNoise $\rightarrow$ SheepNoise baa $\cdot$, $\$$] \}
\[SheepNoise $\rightarrow$ SheepNoise baa $\cdot$, baa] \}

New, but obvious, notation for two distinct items
\{ [SheepNoise $\rightarrow$ baa $\cdot$, $\$$] & [SheepNoise $\rightarrow$ baa $\cdot$, baa] \}

Nothing more to compute, since $\cdot$ is at the end of every item in $cc_3$. 