Syntax Analysis, VI
Examples from LR Parsing

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 3 in EaC2e
Roadmap

Last Class

• Bottom-up parsers, reverse rightmost derivations
• The mystical concept of a handle
  – Easy to understand if we are given an oracle
  – Opaque (at this point) unless we are given an oracle
• Saw a bottom-up, shift-reduce parser at work on \( x - 2 * y \)

This Class

• Structure & operation of an LR(1) parser
  – Both a skeleton parser & the LR(1) tables
• Example from the Parentheses Language
  – Look at how the LR(1) parser uses lookahead to determine shift vs reduce
• Lay the groundwork for the table construction lecture
  – LR(1) items, Closure(), and Goto()
LR(1) Parsers

This week will focus on LR(1) parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 word) for handle recognition
- The class of grammars that these parsers recognize is called the set of LR(1) grammars

Informal definition:
A grammar is LR(1) if, given a rightmost derivation
\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

We can

1. isolate the handle of each right-sentential form \( \gamma_i \), and
2. determine the production by which to reduce,

by scanning \( \gamma_i \) from left-to-right, going at most 1 word beyond the right end of the handle of \( \gamma_i \)

LR(1) implies a left-to-right scan of the input, a rightmost derivation (in reverse), and 1 word of lookahead.
Our conceptual shift-reduce parser from last lecture

```
push INVALID
word ← NextWord( )
repeat until (top of stack = Goal and word = EOF)
    if the top of the stack is a handle A→β
        then  // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (word ≠ EOF)
            then  // shift
                push word
                word ← NextWord( )
        else  // need to shift, but out of input
            report an error
report success
```

Shift-reduce parsers have four kinds of actions:

**Shift:** next word is moved from input to stack

**Reduce:** handle is at **TOS**
- pop **RHS** of handle
- push **LHS** of handle

**Accept:** stop & report success

**Error:** report an error

*Shift & Accept* are \( O(1) \)

*Reduce* is \( O(|RHS|) \) (typically small)

**Key insight:** the parser shifts until a handle appears at **TOS**
The LR(1) Skeleton Parser

The Skeleton LR(1) parser

- follows basic shift-reduce scheme from last slide
- relies on a stack & a scanner
- Stacks \(<\text{symbol, state}>\) pairs
- handle finder is encoded in two tables: ACTION & GOTO
- shifts \(|\text{words}| \) times
- reduces \(|\text{derivation}| \) times
- accepts at most once
- detects errors by failure of the handle-finder, not by exhausting the input

Given tables, we have a parser.

```c
stack.push(INVALID);  // initial state
stack.push(s_0);
word ← NextWord();

loop forever {
    s ← stack.top();
    if ( ACTION[s,word] == "reduce A → β" ) then {
        stack.popnum( 2 * \(|β|\) );  // pop RHS off stack
        s ← stack.top();
        stack.push(A );  // push LHS, A
        stack.push( GOTO[s,A] );  // push next state
    }
    else if ( ACTION[s,word] == "shift s_i" ) then {
        stack.push(word); stack.push( s_i);  
        word ← NextWord();
    }
    else if ( ACTION[s,word] == "accept" & word == EOF) then break;
    else throw a syntax error;
}
report success;
```
The Parentheses Language

Language of Balanced Parentheses

• Any sentence that consists of an equal number of (‘s and )’s
• Beyond the power of regular expressions
  – Classic justification for context-free grammar

1. \textit{Goal} \rightarrow \textit{List}
2. \textit{List} \rightarrow \textit{List \ Pair}
3. \quad | \quad \textit{Pair}
4. \textit{Pair} \rightarrow \{ \textit{List} \}
5. \quad | \quad \{ \}

Good example to elucidate the role of context in \textbf{LR(1)} parsing

This grammar and its tables differ, slightly, from the one in EaC2e.
On Handout

LR(1) Tables for Parenthesis Grammar

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>(</th>
<th>)</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s 3</td>
<td>s 3</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>s₁</td>
<td>s 3</td>
<td></td>
<td></td>
<td>r 3</td>
</tr>
<tr>
<td>s₂</td>
<td>r 3</td>
<td></td>
<td></td>
<td>r 3</td>
</tr>
<tr>
<td>s₃</td>
<td>s 7</td>
<td>s 7</td>
<td>s 8</td>
<td></td>
</tr>
<tr>
<td>s₄</td>
<td>r 2</td>
<td></td>
<td></td>
<td>r 2</td>
</tr>
<tr>
<td>s₅</td>
<td>s 7</td>
<td>s 7</td>
<td>s 10</td>
<td></td>
</tr>
<tr>
<td>s₆</td>
<td>r 3</td>
<td></td>
<td></td>
<td>r 3</td>
</tr>
<tr>
<td>s₇</td>
<td>s 7</td>
<td>s 7</td>
<td>s 12</td>
<td></td>
</tr>
<tr>
<td>s₈</td>
<td>r 5</td>
<td></td>
<td></td>
<td>r 5</td>
</tr>
<tr>
<td>s₉</td>
<td>r 2</td>
<td></td>
<td></td>
<td>r 2</td>
</tr>
<tr>
<td>s₁₀</td>
<td>r 4</td>
<td></td>
<td></td>
<td>r 4</td>
</tr>
<tr>
<td>s₁₁</td>
<td>s 7</td>
<td>s 7</td>
<td>s 13</td>
<td></td>
</tr>
<tr>
<td>s₁₂</td>
<td>r 5</td>
<td></td>
<td></td>
<td>r 5</td>
</tr>
<tr>
<td>s₁₃</td>
<td>r 4</td>
<td></td>
<td></td>
<td>r 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>List</th>
<th>Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>s₁</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>s₂</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>s₃</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>s₄</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>s₅</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>s₆</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₇</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>s₈</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₉</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₀</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₁₃</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“s 23” means shift & goto state 23
“r 18” means reduce by prod’n 18 (& find next state in the GOTO table)
Blank is an error entry
### Parsing "()"

#### The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0 3</td>
<td>—none—</td>
<td>shift 8</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0 3 8</td>
<td>Pair → ( )</td>
<td>reduce 5</td>
</tr>
<tr>
<td>2</td>
<td>EOF</td>
<td>$0 Pair 2</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0 List 1</td>
<td>Goal → List</td>
<td>accept</td>
</tr>
</tbody>
</table>

The **Lookahead** column shows the contents of *word* in the algorithm.
### Parsing "(()())"

The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>$0{3</td>
<td>—none—</td>
<td>shift 7</td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td>$0{3{7</td>
<td>—none—</td>
<td>shift 12</td>
</tr>
<tr>
<td>12</td>
<td>)</td>
<td>$0{3{7}12</td>
<td>Pair $\rightarrow{}</td>
<td>reduce 5</td>
</tr>
<tr>
<td>6</td>
<td>)</td>
<td>$0{3{Pair\ 6</td>
<td>List $\rightarrow\ Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>5</td>
<td>)</td>
<td>$0{3\ List\ 5</td>
<td>—none—</td>
<td>shift 10</td>
</tr>
<tr>
<td>10</td>
<td>)</td>
<td>$0{3\ List\ 5}10</td>
<td>Pair $\rightarrow{\ List}</td>
<td>reduce 4</td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td>$0\ Pair\ 2</td>
<td>List $\rightarrow\ Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>(</td>
<td>$0\ List\ 1</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0\ List\ 1{3</td>
<td>—none—</td>
<td>shift 8</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0\ List\ 1{3}8</td>
<td>Pair $\rightarrow{}</td>
<td>reduce 5</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$0\ List\ 1\ Pair\ 4</td>
<td>List $\rightarrow\ List\ Pair</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0\ List\ 1</td>
<td>Goal $\rightarrow\ List</td>
<td>accept</td>
</tr>
</tbody>
</table>
## Parsing “()

The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>—-none-</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—-none-</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>$0(3)</td>
<td>—-none-</td>
<td>shift 7</td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td>$0(3(7)</td>
<td>—-none-</td>
<td>shift 12</td>
</tr>
<tr>
<td>12</td>
<td>)</td>
<td>$0(3(7)12</td>
<td>—-none-</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>)</td>
<td>$0(3Pair6</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>5</td>
<td>)</td>
<td>$0(3List5</td>
<td>—-none-</td>
<td>shift 10</td>
</tr>
<tr>
<td>10</td>
<td>)</td>
<td>$0(3List510</td>
<td>Pair → List</td>
<td>reduce 4</td>
</tr>
<tr>
<td>2</td>
<td>)</td>
<td>$0Pair2</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>)</td>
<td>$0List1</td>
<td>—-none-</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0List1(3</td>
<td>—-none-</td>
<td>shift 8</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0List1(38</td>
<td>Pair → ( )</td>
<td>reduce 5</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$0List1Pair4</td>
<td>List → List Pair</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0List1</td>
<td>Goal → List</td>
<td>accept</td>
</tr>
</tbody>
</table>
### Parsing “( )”

#### The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>——none——</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>——none——</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0(3)</td>
<td>——none——</td>
<td>shift 8</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0(3)8</td>
<td>Pair → ( )</td>
<td>reduce 5</td>
</tr>
<tr>
<td>2</td>
<td>EOF</td>
<td>$0 Pair 2</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0 List 1</td>
<td>Goal → List</td>
<td>accept</td>
</tr>
</tbody>
</table>

In the string “( )”, reducing by production 5 reveals state $s_0$.

Goto($s_0$, Pair) is $s_2$, which leads to chain of productions 3 & 1.

**Production Rules**

1. Goal → List
2. List → List Pair
3. Pair → ( List )
4. Pair → ( )

---

COMP 412, Fall 2017
Parsing “(()())”

The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>$0 ( 3</td>
<td>—none—</td>
<td>shift 7</td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td>$0 ( 3 ( 7</td>
<td>—none—</td>
<td>shift 12</td>
</tr>
<tr>
<td>12</td>
<td>)</td>
<td>$0 ( 3 ( 7 ) 12</td>
<td>Pair → ( )</td>
<td>reduce 5</td>
</tr>
<tr>
<td>6</td>
<td>)</td>
<td>$0 ( 3 Pair 6</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>5</td>
<td>)</td>
<td>$0 ( 3 List 5</td>
<td>—none—</td>
<td>shift 10</td>
</tr>
<tr>
<td>10</td>
<td>)</td>
<td>$0 ( 3 List 5 ) 10</td>
<td>Pair → { List }</td>
<td>reduce 4</td>
</tr>
</tbody>
</table>

Here, reducing by 5 reveals state $s_3$, which represents the left context of an unmatched ‘(‘. There will be one $s_3$ per unmatched ‘(‘ — they count the remaining ‘(‘s.

Goto($s_3$, Pair) is $s_6$, a state in which the parser expects a ‘)’. That state leads to reductions by 3 and then 4.

1  Goal → List
2  List → List Pair
3  | Pair
4  Pair → ( List )
5  | ( )

COMP 412, Fall 2017
### Parsing "((()))"

#### The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
</tr>
<tr>
<td>1</td>
<td>)</td>
<td>$0 3</td>
<td>reduce 5</td>
</tr>
<tr>
<td>2</td>
<td>)</td>
<td>$0 2</td>
<td>reduce 5</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0 1 ( 3</td>
<td>reduce 3</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$0 1 4</td>
<td>reduce 2</td>
</tr>
<tr>
<td>5</td>
<td>EOF</td>
<td>$0 1</td>
<td>accept</td>
</tr>
</tbody>
</table>

Here, reducing by 5 reveals state $s_1$, which represents the left context of a previously recognized $List$.

Goto($s_1$, $Pair$) is $s_4$, a state in which the parser will reduce $List$ $Pair$ to $List$ (production 2) on a lookahead of either ‘(‘ or $EOF$.

Here, lookahead is $EOF$, which leads to reduction by 2, then by 1.

- $Pair \rightarrow \{ \_list \}$ reduce 5
- $List \rightarrow \pair$ reduce 2
- $Goal \rightarrow \list$ accept
LR(1) Parsers

Recap: How does an LR(1) parser 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 to control the stack-based recognizer
  – ACTION & GOTO tables encode the DFA

• To match a subterm, invoke the DFA recursively
  – leave old DFA’s state on stack and go on

• Final state in DFA ⇒ a reduce action
  – Pop rhs off the stack to reveal invoking state
    → “It would be legal to recognize an x, and we did…”
  – New state is GOTO[revealed state, lhs]
  – Take a DFA transition on the new NT — the LHS we just pushed...
The Control DFA for the parentheses language is embedded in the ACTION and GOTO Tables

→ Transitions on terminals represent shift actions  \[ \text{ACTION Table} \]

→ Transitions on nonterminals follow reduce actions  \[ \text{GOTO Table} \]

The table construction derives this DFA from the grammar.

This point is not obvious. To see it, compare the ACTION & GOTO tables for the parenthesis language with the DFA.
Building LR(1) Tables

How do we generate the ACTION and GOTO tables?

- Use the grammar to build a model of the Control DFA
- Encode actions & transitions in ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)
  - “Succeeds” means defines each table entry uniquely

The Big Picture

- Model the state of the parser with LR(1) items
- Use two functions $\text{goto}(s, X)$ and $\text{closure}(s)$
  - $\text{goto}()$ is analogous to $\text{move}()$ in the subset construction
  - Given a partial state, $\text{closure}()$ adds all the items implied by the partial state
- Build up the states and transition functions of the DFA
- Use this information to fill in the ACTION and GOTO tables

An operational definition

grammar symbol, $T$ or $NT$
goto() is analogous to move() in the subset construction
Given a partial state, closure() adds all the items implied by the partial state

fixed-point algorithm, similar to the subset construction
LR(1) Table Construction

To understand the algorithms, we need to understand the data structure that they use: LR(1) items

• The **LR(1)** table construction algorithm models the set of possible states that the parser can enter
  – Mildly reminiscent of the subset construction (NFA → DFA)

• The construction needs a representation for the parser’s state, as a function of the context it has seen and might see

**LR(1) Items**

• The **LR(1)** table construction algorithm represents each valid configuration of an **LR(1)** parser with an **LR(1)** item

• An **LR(1)** item is a pair \([P, \delta]\), where
  
  * **P** is a production \(A \rightarrow \beta\) with a • at some position in the RHS
  * **\(\delta\)** is a single symbol lookahead

\[\text{(symbol} \equiv \text{word or EOF)}\]
LR(1) Items

An LR(1) item is a pair \([P, \delta]\), where

- \(P\) is a production \(A \rightarrow \beta\) with a • at some position in the RHS
- \(\delta\) is a single symbol lookahead

\((symbol \equiv word\ or\ EOF)\)

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

\([A \rightarrow \bullet \beta\gamma, a]\) means that the input seen so far is consistent with the use of

\(A \rightarrow \beta\gamma\) immediately after the symbol on top of the stack.

We call an item like this a **possibility**.

\([A \rightarrow \beta \bullet \gamma, a]\) means that the input sees so far is consistent with the use of

\(A \rightarrow \beta\gamma\) at this point in the parse, *and* that the parser has already recognized \(\beta\) (that is, \(\beta\) is on top of the stack).

We call an item like this a **partially complete** item.

\([A \rightarrow \beta\gamma \bullet, a]\) means that the parser has seen \(\beta\gamma\), *and* that a lookahead symbol of \(a\) is consistent with reducing to \(A\).

This item is **complete**.

\(\text{LR(k)}\) parsers rely on items with a lookahead of \(\leq k\) symbols.

That leads to LR(k) items, with correspondingly longer \(\delta\).
LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1B_2B_3$ with lookahead $a$, can give rise to 4 items

$$[A \rightarrow \bullet B_1B_2B_3,a]$$
$$[A \rightarrow B_1 \bullet B_2B_3,a]$$
$$[A \rightarrow B_1B_2 \bullet B_3,a]$$
$$[A \rightarrow B_1B_2B_3 \bullet ,a]$$

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

**What’s the point of all these lookahead symbols?**

- Carry them along to help choose the correct reduction
- Lookaheads are bookkeeping, unless item has $\bullet$ at right end
  - Has no direct use in $[A \rightarrow \beta \bullet \gamma,a]$
  - In $[A \rightarrow \beta \bullet,a]$, a lookahead of $a$ implies a reduction by $A \rightarrow \beta$
  - For $\{ [A \rightarrow \beta \bullet,a],[B \rightarrow \gamma \bullet \delta,b] \}$, $a \Rightarrow reduce$ to $A$; FIRST($\delta$) $\Rightarrow$ **shift**

$\Rightarrow$ Limited right context is enough to pick the actions

$$a \in \text{FIRST}(\delta) \Rightarrow a \text{ a conflict, not LR(1)}$$
LR(1) Items: Why should you know this stuff?

Debugging a grammar

- When you build an LR(1) parser, it is possible (likely) that the initial grammar is not LR(1)
- The tools will provide you with debugging output
- To the right is a sample of bison’s output for the if-then-else grammar

The state is described by its LR(1) items
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, $s_0$
      - $[S' \rightarrow S, \text{EOF}]$, 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 Canonical Collection of Sets of LR(1) items

The sets in the canonical collection form the states of the Control DFA.
The construction traces the DFA’s transitions

COMP 412, Fall 2017
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, \( s_0 \)
      - \( [S' \rightarrow S, \text{EOF}] \), 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 Canonical Collection of Sets of LR(1) items

Let’s build the tables for the left-recursive SheepNoise grammar \( (S' \text{ is Goal}) \)

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>→</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SheepNoise</td>
<td>→</td>
<td>SheepNoise, baa</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>→</td>
<td>SheepNoise, baa</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing Closures

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

- Any item \([A \rightarrow \beta \bullet B\delta, a]\) where \(B \in NT\) implies \([B \rightarrow \bullet \tau, x]\) for each production that has \(B\) on the *lhs*, and each \(x \in \text{FIRST}(\delta a)\)
- Since \(\beta B\delta\) is valid, any way to derive \(\beta B\delta\) is valid, too

The Algorithm

**Closure( s )**

```
while ( s is still changing )
    \forall items [A \rightarrow \beta \bullet B\delta, a] \in s
    \forall productions B \rightarrow \tau \in P
    \forall b \in \text{FIRST}(\delta a) \quad // \delta\ might\ be\ \epsilon
    \quad if \quad [B \rightarrow \bullet \tau, b] \notin s
    \quad \quad then \quad s \leftarrow s \cup \{ [B \rightarrow \bullet \tau, b] \}
```

- Classic fixed-point method
- Halts because \(s \subseteq I\), the set of items
- Worklist version is faster
- Closure “fills out” a state \(s\)

Generate new lookaheads. See note on p. 128
Example From SheepNoise

Initial step builds the item \([\text{Goal} \rightarrow \bullet \text{SheepNoise, EOF}]\) and takes its \(\text{Closure( )}\).

\(\text{Closure( [Goal} \rightarrow \bullet \text{SheepNoise, EOF] )}\)

<table>
<thead>
<tr>
<th>Item</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Goal} \rightarrow \bullet \text{SheepNoise, EOF}])</td>
<td>Original item</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, EOF}])</td>
<td>ITER 1, PR 0, (\delta_a) is EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{baa, EOF}])</td>
<td>ITER 1, PR 0, (\delta_a) is EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}])</td>
<td>ITER 2, PR 1, (\delta_a) is baa EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{baa, baa}])</td>
<td>ITER 2, PR 1, (\delta_a) is baa EOF</td>
</tr>
</tbody>
</table>

So, \(S_0\) is

\{ \([\text{Goal} \rightarrow \bullet \text{SheepNoise, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{baa, EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}], [\text{SheepNoise} \rightarrow \bullet \text{baa, baa}] \) \}

This is the left-recursive SheepNoise; EaC2e shows the right-recursive version.
Computing Gotos

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

- *Goto*( { [A→β•Xδ,a] }, *X*) produces [A→βX•δ,a]  

- It finds all such items & uses *Closure()* to fill out the state

The Algorithm

```
Goto( s, X )
new ← Ø
∀ items [A→β•Xδ,a] ∈ s
   new ← new ∪ { [A→βX•δ,a] }
return closure( new )
```

- Not a fixed-point method!
- Straightforward computation
- Uses *Closure()*
- *Goto()* models a transition in the automaton

*Goto* in this construction is analogous to *Move* in the subset construction.
Example from SheepNoise

Assume that \( S_0 \) is

\[
\{ \begin{array}{l}
[\text{Goal} \rightarrow \bullet \text{SheepNoise, EOF}],
[\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, EOF}],
[\text{SheepNoise} \rightarrow \bullet \text{ baa, EOF}],
[\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}],
[\text{SheepNoise} \rightarrow \bullet \text{ baa, baa}] \\
\end{array} \}
\]

\textbf{Goto}( S_0, \text{ baa } )

\begin{itemize}
  \item Loop produces
  \begin{itemize}
    \item \textbf{Item} \hspace{2cm} \textbf{Source}
    \begin{array}{l|l}
      \text{Item 3 in } s_0 & [\text{SheepNoise} \rightarrow \text{ baa } \bullet , \text{ EOF}] \\
      \text{Item 5 in } s_0 & [\text{SheepNoise} \rightarrow \text{ baa } \bullet , \text{ baa}] \\
    \end{array}
  \end{itemize}
  \item \textbf{Closure} adds nothing since \( \bullet \) is at end of \textit{rhs} in each item
\end{itemize}

In the construction, this produces \( s_2 \)

\[
\{[\text{SheepNoise} \rightarrow \text{ baa } \bullet , \{\text{EOF}, \text{ baa}\}]\}
\]

\textbf{From earlier slide}

New, but \textit{obvious}, notation for two distinct items

\[
[\text{SheepNoise} \rightarrow \text{ baa } \bullet , \text{ EOF}] & [\text{SheepNoise} \rightarrow \text{ baa } \bullet , \text{ baa}]
\]

\begin{array}{l|l|l}
0 & \text{Goal} & \rightarrow \text{ SheepNoise} \\
1 & \text{SheepNoise} & \rightarrow \text{ SheepNoise baa} \\
2 & | & \text{ baa}
\end{array}

COMP 412, Fall 2017
Building the Canonical Collection

Start from $s_0 = \text{Closure}( [S' \rightarrow S, EOF] )$

Repeatedly construct new states, until all are found

The Algorithm

\[
s_0 \leftarrow \text{Closure}( [S' \rightarrow S, EOF] ) \\
S \leftarrow \{ s_0 \} \\
k \leftarrow 1 \\
\text{while (} S \text{ is still changing) }
\}
\forall s_j \in S \text{ and } \forall x \in ( T \cup NT ) \\
s_k \leftarrow \text{Goto}(s_j, x) \\
\text{record } s_j \rightarrow s_k \text{ on } x \\
\text{if } s_k \not\in S \text{ then} \\
S \leftarrow S \cup \{ s_k \} \\
k \leftarrow k + 1
\]

- Fixed-point computation
- Loop adds to $S$
- $S \subseteq 2^{\text{ITEMS}}$, so $S$ is finite
- Worklist version is faster because it avoids duplicated effort

This membership / equality test requires careful and/or clever implementation.
Example from SheepNoise

Starts with $S_0$

$S_0 : \{ [\text{Goal} \rightarrow \text{SheepNoise} \cdot, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa}, \text{EOF}],$

$\quad [\text{SheepNoise} \rightarrow \text{ baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa}, \text{ baa}],$

$\quad [\text{SheepNoise} \rightarrow \text{ baa}, \text{ baa}] \}$

Iteration 1 computes

$S_1 = \textit{Goto}(S_0, \text{SheepNoise}) =$

$\quad \{ [\text{Goal} \rightarrow \text{SheepNoise} \cdot, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \cdot \text{ baa}, \text{EOF}],$

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

$S_2 = \textit{Goto}(S_0, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{ baa} \cdot, \text{EOF}],$

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

Iteration 2 computes

$S_3 = \textit{Goto}(S_1, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \cdot, \text{EOF}],$

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

Nothing more to compute, since $\cdot$ is at the end of every item in $S_3$. 

<table>
<thead>
<tr>
<th>0</th>
<th>Goal</th>
<th>$\rightarrow$</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SheepNoise</td>
<td>$\rightarrow$</td>
<td>SheepNoise baa</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>baa</td>
<td></td>
</tr>
</tbody>
</table>
Example from SheepNoise

\[ S_0 : \{ [\text{Goal} \rightarrow \bullet \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa}, \text{EOF}], \\
[\text{SheepNoise} \rightarrow \bullet \text{ baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa}], \\
[\text{SheepNoise} \rightarrow \bullet \text{ baa, baa}] \} \]

\[ S_1 = \text{Goto}(S_0, \text{SheepNoise}) = \\
\{ [\text{Goal} \rightarrow \text{SheepNoise} \bullet, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa}, \text{EOF}], \\
[\text{SheepNoise} \rightarrow \text{SheepNoise} \bullet \text{ baa, baa}] \} \]

\[ S_2 = \text{Goto}(S_0, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{ baa} \bullet, \text{EOF}], [\text{SheepNoise} \rightarrow \text{ baa} \bullet, \text{ baa}] \} \]

\[ S_3 = \text{Goto}(S_1, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \bullet, \text{EOF}], \\
[\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \bullet, \text{ baa}] \} \]