Midterm Exam: Thursday
October 18, 7PM
Herzstein Amphitheater

Syntax Analysis, VI
Examples from LR Parsing

Comp 412

Copyright 2018, 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 \( x - 2 \times y \)

This Class

• Structure & operation of an \( \text{LR}(1) \) parser
  – Both a skeleton parser & the \( \text{LR}(1) \) tables
• Example from the Parentheses Language
  – Look at how the \( \text{LR}(1) \) parser uses lookahead to determine \textit{shift vs reduce}
• Lay the groundwork for the table construction lecture
  – \( \text{LR}(1) \) items, \textit{Closure()}, and \textit{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.
Bottom-up Parser

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 forms a handle $A \rightarrow \beta$ then
    // reduce $\beta$ to $A$
    pop $|\beta|$ symbols off the stack
    push $A$ onto the stack
  else if (word ≠ EOF) then
    // shift
    push word
    word ← NextWord()
  else
    // error: out of input
    report an error
report success  // accept
```

Shift-reduce parsers have four kinds of actions:

**Shift:** move next word from input to the 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 $O(|\text{RHS}|)$

\[ \sum |\text{RHS}| = |\text{Parse tree nodes}| \]

Key insight: the parser shifts until a handle appears at TOS
Table-Driven LR Parsers

A table-driven LR(1) parser is a bottom-up shift-reduce parser

- Grammatical knowledge is encoded in two tables: Action & Goto
  - They encode the handle-finding automaton
  - They are constructed by an LR(1) parser generator

- Why two tables?
  - Reduce needs more information & more complex information than shift
  - Goto holds that extra information
The LR(1) Skeleton Parser

Given tables, we have a parser.

```c
stack.push(INVALID); stack.push(s0);          // initial state
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 si”) then {
        stack.push(word); stack.push(si);
        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. \( Goal \rightarrow List \)
2. \( List \rightarrow List \; \text{Pair} \)
3. \( \vert \; \text{Pair} \)
4. \( Pair \rightarrow \{ \; List \; \} \)
5. \( \vert \; \{ \; \} \)

Good example to elucidate the role of context in LR(1) parsing

This grammar and its tables differ, slightly, from the one in EaC2e.
### LR(1) Tables for Parenthesis Grammar

#### On Handout

<table>
<thead>
<tr>
<th>ACTION</th>
<th>State</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s0</td>
<td>s3</td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>s3</td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>r3</td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>s7</td>
</tr>
<tr>
<td></td>
<td>s4</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>s5</td>
<td>s7</td>
</tr>
<tr>
<td></td>
<td>s6</td>
<td>r3</td>
</tr>
<tr>
<td></td>
<td>s7</td>
<td>s7</td>
</tr>
<tr>
<td></td>
<td>s8</td>
<td>r5</td>
</tr>
<tr>
<td></td>
<td>s9</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>s10</td>
<td>r4</td>
</tr>
<tr>
<td></td>
<td>s11</td>
<td>s7</td>
</tr>
<tr>
<td></td>
<td>s12</td>
<td>r5</td>
</tr>
<tr>
<td></td>
<td>s13</td>
<td>r4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GOTO</th>
<th>State</th>
<th>List</th>
<th>Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>s4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s5</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s7</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. **Goal** \rightarrow **List**
2. **List** \rightarrow **List Pair**
3. **List Pair** \rightarrow **Pair**
4. **Pair** \rightarrow (** List **)  

- "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>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>Look-ahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(</td>
<td>$0$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>$0 (3$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td>$0 (3 (7$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>12</td>
<td>)</td>
<td>$0 (3 (7 )12</td>
<td></td>
<td></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>—</td>
</tr>
<tr>
<td>10</td>
<td>)</td>
<td>$0 (3 List 5)10</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>)</td>
<td>$0 Pair 2</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>)</td>
<td>$0 List 1</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0 List 1 (3</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0 List 1 (3 )8</td>
<td></td>
<td>reduce 5</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$0 List 1 Pair 4</td>
<td></td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0 List 1</td>
<td>Goal → List</td>
<td>accept</td>
</tr>
</tbody>
</table>

**Production Rules:**

- Goal → List
- List → List Pair
- List → Pair
- Pair → ( List )
### Parsing “((())())”

#### The Parentheses Language

<table>
<thead>
<tr>
<th>State</th>
<th>Look-ahead</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 ( 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>
<tr>
<td>2</td>
<td>)</td>
<td>$ 0 Pair 2</td>
<td>List → 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 → ( )</td>
<td>reduce 5</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$ 0 List 1 Pair 4</td>
<td>List → List Pair</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$ 0 List 1</td>
<td>Goal → List</td>
<td>accept</td>
</tr>
</tbody>
</table>

Let’s look at how it reduces “(())”. We have seen 3 examples.
## 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>—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>$0 ( 3</td>
<td>—</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.

```
1  Goal   –>   List
2  List   –>   List Pair
3  |       –>   Pair
4  Pair   –>   ( List )
5  |       –>   ( )
```
## 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 that must be matched.

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. | ( List )
5. | ( )
### 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>3</td>
<td>)</td>
<td>$0</td>
<td>shift 3</td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td>$0</td>
<td>shift 7</td>
</tr>
<tr>
<td>12</td>
<td>)</td>
<td>$0</td>
<td>shift 12</td>
</tr>
<tr>
<td>10</td>
<td>)</td>
<td>$0</td>
<td>reduce 5</td>
</tr>
<tr>
<td>2</td>
<td>)</td>
<td>$0</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>)</td>
<td>$0</td>
<td>reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0</td>
<td>reduce 3</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0</td>
<td>reduce 4</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$0</td>
<td>reduce 2</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0</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.

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>EOF</td>
<td>$0</td>
<td>accept</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0</td>
<td>accept</td>
</tr>
</tbody>
</table>

1. **Goal** $\rightarrow$ List
2. **List** $\rightarrow$ List Pair
3. List $\mid$ Pair
4. **Pair** $\rightarrow$ ( List )
5. **List** $\rightarrow$ Pair
6. **Pair** $\rightarrow$ ( )
7. **Pair** $\rightarrow$ ( List )
8. **Pair** $\rightarrow$ ( )
9. **List** $\rightarrow$ List Pair
10. **List** $\rightarrow$ List
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
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.
**LR(1) Parsers**

**Recap: How does an LR(1) parser work?**

- Unambiguous grammar $\Rightarrow$ 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, it, effectively, invokes the DFA recursively
  - leave old DFA’s state on stack and go on
- Final state in DFA $\Rightarrow$ a *reduce* action  (& a return from the “recursion”)
  - 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, NT on LHS]
  - Take a DFA transition on the new NT — the LHS we just pushed...
Building \textbf{LR(1)} Tables

How do we generate the ACTION and GOTO tables?

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

The Big Picture

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

\[ \text{An operational definition} \]
\[ \text{grammar symbol, } T \text{ or } NT \]
\[ \text{fixed-point algorithm, similar to the subset construction} \]
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 ($symbol \equiv word$ 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

\([A \rightarrow •\beta\gamma, a]\) 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**.

\([A \rightarrow \beta • \gamma, a]\) 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.

\([A \rightarrow \beta\gamma •, a]\) 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**.

**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 $\underline{a}$, can give rise to 4 items

$$[A \rightarrow \bullet B_1B_2B_3, \underline{a}], [A \rightarrow B_1 \bullet B_2B_3, \underline{a}], [A \rightarrow B_1B_2 \bullet B_3, \underline{a}], \text{ & } [A \rightarrow B_1B_2B_3 \bullet, \underline{a}]$$

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 $[A \rightarrow \beta \bullet \gamma, \underline{a}]$
  - In $[A \rightarrow \beta \bullet, \underline{a}]$, a lookahead of $\underline{a}$ implies a reduction by $A \rightarrow \beta$
  - For $\{ [A \rightarrow \beta \bullet, \underline{a}], [B \rightarrow \gamma \bullet \delta, \underline{b}] \}$, $\underline{a} \Rightarrow \text{reduce to } A; \text{ FIRST}(\delta) \Rightarrow \text{shift}$

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

```
  goal    :  stmt_list  
  stmt_list :  stmt_list stmt 
              |  stmt  
  stmt    :  IF_EXPR THEN stmt  
          |  IF_EXPR THEN stmt  ELSE stmt  
          |  OTHER  
```

The state is described by its LR(1) items

state 10

```
  4  stmt : IF_EXPR THEN stmt .  
  5  | IF_EXPR THEN stmt . ELSE stmt  
  ELSE  shift, and go to state 11  
  ELSE  [reduce using rule 4 (stmt)]  
        $default  reduce using rule 4 (stmt)
```

That period is the •

Likely STOP
**LR(1) Table Construction**

**High-level overview**

1. Build the Canonical Collection of Sets of LR(1) Items, \( I \)
   - Begin in an appropriate state, \( s_0 \)
     - \( [S' \rightarrow \cdot S, \text{EOF}] \), 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 \( \text{goto}( ) \)
   - 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
# 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, \( s_0 \)
      - \( [S' \rightarrow \ast S, \text{EOF}] \), 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 \( (S' \text{ is Goal}) \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>( \rightarrow ) SheepNoise</td>
</tr>
<tr>
<td>1</td>
<td>SheepNoise</td>
<td>( \rightarrow ) SheepNoise baa</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>( \mid ) baa</td>
</tr>
</tbody>
</table>

COMP 412, Fall 2018
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 \(x \in FIRST(\delta a)\)
- 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)

\[\forall items [A \rightarrow \beta \bullet B\delta, a] \in s\]
\[\forall productions B \rightarrow \tau \in P\]
\[\forall b \in FIRST(\delta a) \quad // \delta\ might\ be\ \varepsilon\]

if \([B \rightarrow \bullet \tau, b] \notin s\)

then \(s \leftarrow s \cup \{ [B \rightarrow \bullet \tau, b] \}\)

- 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
Computing Closures

Generating Closures is the place where a human is most likely to make a mistake

- With everything going on in the construction, it is easy to lose track of δa and the fact that it refers to the item, not the current production

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

- The lookahead computation is a great example of why these table constructions should be done by computers, not human beings
This is the *left-recursive* SheepNoise; EaC2e shows the *right-recursive* version.

**Example From SheepNoise**

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

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

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

So, \(S_0\) is

\[
\{ \left[ \text{Goal} \rightarrow \bullet \text{SheepNoise}, \text{EOF} \right], \left[ \text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa}, \text{EOF} \right],
\left[ \text{SheepNoise} \rightarrow \bullet \text{ baa}, \text{EOF} \right], \left[ \text{SheepNoise} \rightarrow \bullet \text{SheepNoise baa, baa} \right],
\left[ \text{SheepNoise} \rightarrow \bullet \text{ baa, baa} \right] \}
\]
Computing Gotos

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

- \( Goto( \{ [A \rightarrow \beta \cdot X \delta, a] \}, X ) \) produces \( [A \rightarrow \beta X \cdot \delta, a] \) \( \text{ (obviously) } \)
- It finds all such items & uses \( \text{Closure}() \) to fill out the state

The Algorithm

\[
Goto( s, X ) \\
\text{ new } \leftarrow \emptyset \\
\forall \text{ items } [A \rightarrow \beta \cdot X \delta, a] \in s \\
\text{ new } \leftarrow \text{ new } \cup \{ [A \rightarrow \beta X \cdot \delta, a] \} \\
\text{ return } \text{ closure}( \text{ new } )
\]

- Not a fixed-point method!
- Straightforward computation
- Uses \( \text{Closure}( ) \)
- \( Goto( ) \) models a transition in the automaton

\( Goto \) in this construction is analogous to \( \text{Move} \) in the subset construction.
Example from SheepNoise

Assume that $S_0$ is

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

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

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

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

• Loop produces

<table>
<thead>
<tr>
<th>Item</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SheepNoise $\rightarrow$ baa $\bullet$, EOF]</td>
<td>Item 3 in $s_0$</td>
</tr>
<tr>
<td>[SheepNoise $\rightarrow$ baa $\bullet$, baa]</td>
<td>Item 5 in $s_0$</td>
</tr>
</tbody>
</table>

• \textbf{Closure} adds nothing since $\bullet$ is at end of rhs in each item

In the construction, this produces $s_2$

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

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

$[\text{SheepNoise} \rightarrow \text{baa} \bullet, \text{EOF}]$ & $[\text{SheepNoise} \rightarrow \text{baa} \bullet, \text{baa}]$
Building the Canonical Collection

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

Repeatedly construct new states, until all are found

The Algorithm

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

- 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 $S_0$

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

Iteration 1 computes

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

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

Iteration 2 computes

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

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

COMP 412, Fall 2018
Example from SheepNoise

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

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

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

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