Syntax Analysis, VI

Examples from LR Parsing

Comp 412

This lecture is the last one covered on the first exam.
Roadmap

Last Class

• Bottom-up parsers, reverse rightmost derivations
• The mystical concept of a handle
  – Easy to understand if we are given an oracle
  – Can be made efficient if we reformulate position to be stack-relative
• 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 forms 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  // 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 $O(|RHS|)$

$\Sigma |RHS| = |$ Parse tree nodes $|$
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
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

- follows basic shift-reduce scheme from last slide
- relies on a stack & a scanner
- Stacks <symbol, state> pairs
- handle finder is encoded in two tables: ACTION & GOTO
- shifts |words| times
- reduces |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.

EaC2e shows the <symbol, state> pairs as individual elements on the stack.
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  →  List
2. List  →  List  Pair
3.      |  Pair
4. Pair  →  ( List )
5.      |  ()
```

Good example to elucidate the role of context in LR(1) parsing
# LR(1) Tables for Parenthesis Grammar

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>S₀</td>
<td>s 3</td>
<td>1</td>
</tr>
<tr>
<td>S₁</td>
<td>s 3</td>
<td>2</td>
</tr>
<tr>
<td>S₂</td>
<td>r 3</td>
<td>3</td>
</tr>
<tr>
<td>S₃</td>
<td>s 7</td>
<td>4</td>
</tr>
<tr>
<td>S₄</td>
<td>r 2</td>
<td>5</td>
</tr>
<tr>
<td>S₅</td>
<td>s 7</td>
<td>6</td>
</tr>
<tr>
<td>S₆</td>
<td>r 3</td>
<td>7</td>
</tr>
<tr>
<td>S₇</td>
<td>s 7</td>
<td>8</td>
</tr>
<tr>
<td>S₈</td>
<td>r 5</td>
<td>9</td>
</tr>
<tr>
<td>S₉</td>
<td>r 2</td>
<td>10</td>
</tr>
<tr>
<td>S₁₀</td>
<td>r 4</td>
<td>11</td>
</tr>
<tr>
<td>S₁₁</td>
<td>s 7</td>
<td>12</td>
</tr>
<tr>
<td>S₁₂</td>
<td>r 5</td>
<td>13</td>
</tr>
<tr>
<td>S₁₃</td>
<td>r 4</td>
<td>2</td>
</tr>
</tbody>
</table>

**ACTION**

- “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>$Pair \rightarrow (,)$</td>
<td>reduce 5</td>
</tr>
<tr>
<td>2</td>
<td>EOF</td>
<td>$0,Pair,2$</td>
<td>$List \rightarrow Pair$</td>
<td>reduce 3</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>

The **Lookahead** column shows the contents of *word* in the algorithm.

---

COMP 412, Fall 2019
## 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>Pair → (</td>
<td>reduce 5</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>$0(3List5)$</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>$0List1(3)$</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>

Goal → List
List → 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></td>
<td>reduce 5</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(3List5)10</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(3)8</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>

Let’s look at how it reduces "()". We have seen 3 examples.
In the string "( )", reducing by production 5 reveals state $s_0$.

$\text{Goto}(s_0, \text{Pair})$ is $s_2$, which leads to chain of productions 3 & 1.
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 $0</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>$0 $0 $3</td>
<td>—none—</td>
<td>shift 7</td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td>$0 $0 $3 $7</td>
<td>—none—</td>
<td>shift 12</td>
</tr>
<tr>
<td>12</td>
<td>)</td>
<td>$0 $0 $3 $7 $12</td>
<td>Pair → ( )</td>
<td>reduce 5</td>
</tr>
<tr>
<td>6</td>
<td>)</td>
<td>$0 $0 $3 $7 $12</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>5</td>
<td>)</td>
<td>$0 $0 $3 $7 $12</td>
<td>—none—</td>
<td>reduce 4</td>
</tr>
<tr>
<td>10</td>
<td>)</td>
<td>$0 $0 $3 $7 $12</td>
<td>Pair → { List}</td>
<td>reduce 3</td>
</tr>
<tr>
<td></td>
<td>EOF</td>
<td>$0 $0 $3 $7 $12</td>
<td>—</td>
<td>shift 3</td>
</tr>
</tbody>
</table>

Here, reducing by 5 reveals state s₃, which represents the left context of an unmatched ‘(’. There will be one s₃ per unmatched ‘(‘ — they count the remaining ‘(‘s that must be matched.

Goto(s₃, Pair) is s₆, a state in which the parser expects a ‘)’. That state leads to reductions by 3 and then 4.
### 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>10</td>
<td>(</td>
<td>$ 0 ( 3 List 5 ) 10</td>
<td>Pair → ( List )</td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td>$ 0 Pair 2</td>
<td>List → Pair</td>
</tr>
<tr>
<td>1</td>
<td>(</td>
<td>$ 0 List 1</td>
<td>—none—</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$ 0 List 1 ( 3</td>
<td>—none—</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$ 0 List 1 ( 3 ) 8</td>
<td>Pair → ( )</td>
</tr>
<tr>
<td>4</td>
<td>EOF</td>
<td>$ 0 List 1 Pair 4</td>
<td>List → List Pair</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$ 0 List 1</td>
<td>Goal → List</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.
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 automaton for the parentheses language is embedded in the ACTION and GOTO Tables

→ Transitions on **terminals** represent shift actions  [ACTION Table]
→ Transitions on **nonterminals** follow reduce actions  [GOTO Table]

The table construction derives this automaton from the grammar.

This point is not obvious. To see it, compare the ACTION & GOTO tables for the parenthesis language with the automaton.
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, it, effectively, invokes the DFA recursively
  – leave old DFA’s state on stack and go on

• Final state in DFA ⇒ 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 \( LR(1) \) 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 \( goto(s, X) \) and \( closure(s) \)
  - \( goto() \) is analogous to \( move() \) in the subset construction
  - Given a partial state, \( closure() \) 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
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

The intermediate representation of the LR(1) table construction algorithm

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 seen 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**.

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

$$[A \rightarrow \quad B_1B_2B_3,a], [A \rightarrow B_1 \quad B_2B_3,a], [A \rightarrow B_1B_2 \quad B_3,a], \text{ and } [A \rightarrow B_1B_2B_3, \quad a]$$

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 $A$; $\text{FIRST}(\delta) \Rightarrow shift$

$\Rightarrow$ 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

```
goal        : stmt_list
stmt_list   : stmt_list stmt
|           : stmt
stmt        : IF EXPR THEN stmt
|           : IF EXPR THEN stmt ELSE stmt
|           : OTHER
```

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)
LR(1) Table Construction

High-level overview

1. Build the Canonical Collection of Sets of LR(1) Items, $I$
   - Begin in an appropriate state, $s_0$
     - $[S' \rightarrow \bullet S, \text{EOF}]$, 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, $s_0$
     - $[S' \rightarrow S, \text{EOF}]$, 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**

$(S’$ is $\text{Goal})$

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

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

- 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

COMP 412, Fall 2019 25
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 $\delta a$ and the fact that it refers to the item, not the current production.

**Closure( s )**

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

- The lookahead computation is a great example of why these table constructions should be done by computers, not human beings.

In our experience, this use of `FIRST($\delta a$)` is the point in the process where a human is most likely to make a mistake.
Example From SheepNoise

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

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

<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, EOF}]</td>
<td>ITER 1, PR 1, (\delta a \text{ is EOF})</td>
</tr>
<tr>
<td>[\text{SheepNoise} \rightarrow \bullet \text{ baa, EOF}]</td>
<td>ITER 1, PR 2, (\delta a \text{ is EOF})</td>
</tr>
<tr>
<td>[\text{SheepNoise} \rightarrow \bullet \text{SheepNoise} \text{ baa, baa}]</td>
<td>ITER 2, PR 1, (\delta a \text{ is baa EOF})</td>
</tr>
<tr>
<td>[\text{SheepNoise} \rightarrow \bullet \text{ baa, baa}]</td>
<td>ITER 2, PR 2, (\delta a \text{ is baa EOF})</td>
</tr>
</tbody>
</table>

So, \(S_0\) is

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

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

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

The Algorithm

```
Goto( s, X )
new ← ∅
∀ 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

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

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

• Loop produces

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

• $\text{Closure}$ 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 obvious, notation for two distinct items

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

0 | Goal | $\rightarrow$ | SheepNoise |
1 | SheepNoise | $\rightarrow$ | SheepNoise baa |
2 | | $\mid$ baa |
Building the Canonical Collection

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

Repeatedly construct new states, until all are found

The Algorithm

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

- 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 $S_0$**

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

**Iteration 1 computes**

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

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

**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 $S_3$. 

| 0 | Goal | $\rightarrow$ | SheepNoise |
| 1 | SheepNoise | $\rightarrow$ | SheepNoise baa |
| 2 | | $|$ | baa |
Example from SheepNoise

$S_0 : \{ \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]} \} $

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