These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Homework 5 has been posted. Covers a recursive descent parser and code generator. Due: Monday, April 2

• Homework 6 will be posted by Friday, March 29
Parsing
(Syntax Analysis)

EAC Chapters 3.4
**LR(1), operator precedence**

- 1 input symbol lookahead
- Construct rightmost derivation (backwards)
- Input: read left-to-right

**Rule** \( B \rightarrow \gamma \)

\[
S \Rightarrow_{\text{rm}}^* \alpha B y \Rightarrow_{\text{rm}} \alpha \gamma y \Rightarrow_{\text{rm}}^* \alpha x y
\]
Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

1 input symbol lookahead
construct rightmost derivation (backwards)
input: read left-to-right

input: read left-to-right

upper fringe

\[ S \Rightarrow \alpha B y \Rightarrow \alpha \gamma y \Rightarrow x y \]

rule \( B \rightarrow \gamma \)
Finding Reductions  (Handles)

The parser must find a substring $\beta$ of the tree's frontier that matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation.

Informally, we call this substring $\beta$ a *handle*.

Formally,

A *handle* of a right-sentential form $\gamma$ is a pair $<A \rightarrow \beta,k>$ where $A \rightarrow \beta \in P$ and $k$ is the position in $\gamma$ of $\beta$'s rightmost symbol.

If $<A \rightarrow \beta,k>$ is a handle, then replacing $\beta$ at $k$ with $A$ produces the right sentential form from which $\gamma$ is derived in the rightmost derivation.

Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

$\Rightarrow$ the parser doesn't need to scan past the handle (only lookahead).

$\Rightarrow$ The right end of the handle will be on top of the stack, not within the stack. Need lookahead to determine whether we reached the handle.
Critical Insight

If $G$ is unambiguous, then every right-sentential form has a unique handle.

If we can find those handles, we can build a derivation!

Sketch of Proof:
1. $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to derive $\gamma_i$ from $\gamma_{i-1}$
3. $\Rightarrow$ a unique position $k$ at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $<A \rightarrow \beta, k>$

This all follows from the definitions
The expression grammar handles for rightmost derivation of \( x - 2 * y \)
The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*.

Handle pruning forms the basis for a bottom-up parsing method.

To construct a rightmost derivation
\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w \]

Apply the following simple algorithm

for \( i \gets n \) to 1 by -1

*Find the handle \( \langle A_i \rightarrow \beta_i, k_i \rangle \) in \( \gamma_i \)*

*Replace \( \beta_i \) with \( A_i \) to generate \( \gamma_{i-1} \)*

This takes \( 2n \) steps.
One implementation technique is the *shift-reduce parser*

```plaintext
push INVALID // bottom of stack marker
token ← next_token()
repeat until (top of stack = Goal and token = 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 (token ≠ EOF)
    then // shift
      push token
      token ← next_token()
  else // need to shift but out of input
    report an error
```

**How do errors show up?**
- failure to find a handle
- hitting EOF & needing to shift (final else clause)

Either generates an error

*Figure 3.7 in EAC*
### 1. Shift until the top of the stack is the right end of a handle

### 2. Find the left end of the handle & reduce

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td></td>
<td>none</td>
<td>shift</td>
</tr>
</tbody>
</table>
| $ \text{id} | \text{id} = \text{num} * \text{id}  
             | $\text{num} * \text{id}$           |        |        |
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

<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></td>
<td></td>
</tr>
</tbody>
</table>
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

<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 -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - num</td>
<td>* id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>Input</td>
<td>Handle</td>
<td>Action</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<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 -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - num</td>
<td>* 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></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
# Back to $x = 2 \cdot y$

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

<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>$Expr -$</td>
<td>$num -$ id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr -$ num</td>
<td>$*$ 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>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr -$ Term *</td>
<td>$-$ id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr -$ Term * id</td>
<td>$id</td>
<td>none</td>
<td>shift</td>
</tr>
</tbody>
</table>
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

<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 -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - num</td>
<td>* 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>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - Term *</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr - Term * id</td>
<td></td>
<td>9,5</td>
<td>red. 9</td>
</tr>
<tr>
<td>$ Expr - Term * Factor</td>
<td></td>
<td>5,5</td>
<td>red. 5</td>
</tr>
<tr>
<td>$ Expr - Term</td>
<td></td>
<td>3,3</td>
<td>red. 3</td>
</tr>
<tr>
<td>$ Expr</td>
<td></td>
<td>1,1</td>
<td>red. 1</td>
</tr>
<tr>
<td>$ Goal</td>
<td></td>
<td>none</td>
<td>accept</td>
</tr>
</tbody>
</table>
### 1. Shift until the top of the stack is the right end of a handle

### 2. Find the left end of the handle & reduce

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

Back to $x = 2 \times y$

Diagram:

- Goal
  - Expr
    - Term
      - Fact.
        - <id,y>
      - <num,2>
    - Fact.
      - <id,x>
      - <num,2>
LR(1) Skeleton Parser

```
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 $A \rightarrow \beta$" ) then {
        stack.popnum(2*$\mid \beta \mid$); // pop $2*\mid \beta \mid$ 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 == EOF )
        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
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 $\text{goto}(s, X)$ and $\text{closure}(s)$
  $\rightarrow$ $\text{goto}()$ is analogous to $\text{move}()$ in the subset construction
  $\rightarrow$ $\text{closure}()$ 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
The LR(1) table construction algorithm uses LR(1) items to represent valid configurations of an LR(1) parser.

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

- \(P\) is a production \(A \rightarrow \beta\) with a \(\cdot\) at some position in the rhs
- \(\delta\) is a lookahead string of length \(\leq k\) (words or EOF)

The \(\cdot\) in an item indicates the position of the top of the stack.

**LR(1):**

- \([A \rightarrow \cdot \beta \gamma, a]\) 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.
- \([A \rightarrow \beta \cdot \gamma, a]\) 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 \(\beta\).
- \([A \rightarrow \beta \gamma \cdot, a]\) means that the parser has seen \(\beta \gamma\), and that a lookahead symbol of \(a\) is consistent with reducing to \(A\).
LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1 B_1 B_1$ with lookahead $a$, can give rise to 4 items

\[ [A \rightarrow \cdot B_1 B_2 B_3, a], [A \rightarrow B_1 \cdot B_2 B_3, a], [A \rightarrow B_1 B_2 \cdot B_3, a], \& [A \rightarrow B_1 B_2 B_3 \cdot, a] \]

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 the right end
  - Has no direct use in $[A \rightarrow \beta \cdot \gamma, a]$
  - In $[A \rightarrow \beta \cdot, a]$, a lookahead of $a$ implies a reduction by $A \rightarrow \beta$
  - For $\{ [A \rightarrow \beta \cdot, a], [B \rightarrow \gamma \cdot c, b] \}$, $a \Rightarrow \text{reduce to } A$; $c \Rightarrow \text{shift}$
  - Limited right context is enough to pick the actions (unique, i.e., deterministic choice)
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)
     - \( [S' \rightarrow \cdot S, \text{EOF}] \), 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 collection of sets of LR(1) items**

   The canonical collection completely encodes the transition diagram for the handle-finding DFA
Computing Closures

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

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

The algorithm

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

- Classic fixed-point method
- Halts because \(s \subseteq \text{ITEMS}\)
- Worklist version is faster

Closure “fills out” a state
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 \([A \rightarrow \beta X \cdot \delta, a]\) \hspace{1cm} (easy part)
- Should also includes \textit{closure(}[A \rightarrow \beta X \cdot \delta, a]\) \hspace{1cm} (fill out the state)

The algorithm

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

- Not a fixed-point method!
- Straightforward computation
- Uses \textit{closure( )}

\( \text{Goto()} \) moves forward
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

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

- Fixed-point computation (worklist version)
- Loop adds to $CC$
- $CC \subseteq 2^{\text{ITEMS}}$, so $CC$ is finite
The SheepNoise Grammar

We will use this grammar:

1: Goal $\rightarrow$ SheepNoise
2: SheepNoise $\rightarrow$ SheepNoise $\text{baa}$
3: $\text{baa}$
Example From SheepNoise

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

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

<table>
<thead>
<tr>
<th>Item</th>
<th>From</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 \ baa}, \text{EOF}])</td>
<td>1, (\delta_a) is EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{ baa}, \text{EOF}])</td>
<td>1, (\delta_a) is EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{SheepNoise \ baa}, \text{baa}])</td>
<td>2, (\delta_a) is baa EOF</td>
</tr>
<tr>
<td>([\text{SheepNoise} \rightarrow \bullet \text{ baa}, \text{baa}])</td>
<td>2, (\delta_a) is baa EOF</td>
</tr>
</tbody>
</table>

So, \(cc_0\) is

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

\[ c_{c_0} \text{ is } \{ [\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \cdot \text{baa}, \text{EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa, baa}], [\text{SheepNoise} \rightarrow \cdot \text{baa, baa}] \} \]

\[ \text{Goto}(c_{c_0}, \text{baa}) \]
- Loop produces

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

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

In the construction, this produces \(c_{c_2}\)
\[
\{ [\text{SheepNoise} \rightarrow \text{baa} \cdot, \{\text{EOF}, \text{baa}\}] \}
\[
\text{New, but obvious, notation for two distinct items}\]
\[
[\text{SheepNoise} \rightarrow \text{baa} \cdot, \text{EOF}] \& [\text{SheepNoise} \rightarrow \text{baa} \cdot, \text{baa}]\]
Example from SheepNoise

Starts with $cc_0$

$cc_0 : \{ [Goal \rightarrow \cdot \text{SheepNoise, EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise} \ baa, \ EOF],$
$[\text{SheepNoise} \rightarrow \cdot \ baa, \ EOF], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise} \ baa, \ baa],$
$[\text{SheepNoise} \rightarrow \cdot \ baa, \ baa] \}$
Example from SheepNoise

Starts with $cc_0$

$$cc_0: \{ [\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise baa}, \text{EOF}],$$

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

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

Iteration 1 computes

$$cc_1 = \text{Goto}(cc_0, \text{SheepNoise}) =$$

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

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

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

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

Starts with $cc_0$

$$cc_0: \{ [\text{Goal} \rightarrow \cdot \text{SheepNoise}, \text{EOF}], [\text{SheepNoise} \rightarrow \cdot \text{SheepNoise} \cdot \text{baa}, \text{EOF}],$$

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

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

Iteration 1 computes

$$cc_1 = \text{Goto}(cc_0, \text{SheepNoise}) =$$

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

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

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

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

Iteration 2 computes

$$cc_3 = \text{Goto}(cc_1, \text{baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise} \cdot \text{baa}, \text{EOF}],$$

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

Starts with $cc_0$

$cc_0 : \{ [Goal \rightarrow \cdot \ SheepNoise, \ EOF], [SheepNoise \rightarrow \cdot \ SheepNoise \ baa, \ EOF],$

$[SheepNoise \rightarrow \cdot \ baa, \ EOF], [SheepNoise \rightarrow \cdot \ SheepNoise \ baa, \ baa],$

$[SheepNoise \rightarrow \cdot \ baa, \ baa] \}$

Iteration 1 computes

$cc_1 = Goto(cc_0, \ SheepNoise) =$

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

$[SheepNoise \rightarrow \ SheepNoise \cdot \ baa, \ baa] \}$

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

$[SheepNoise \rightarrow \ baa \cdot, \ baa] \}$

Iteration 2 computes

$cc_3 = Goto(cc_1, \ baa) = \{ [SheepNoise \rightarrow \ SheepNoise \ baa \cdot, \ EOF],$

$[SheepNoise \rightarrow \ SheepNoise \ baa \cdot, \ baa] \}$

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

Canonical LR(1) Collection:

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

Iteration 1 computes

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

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

Iteration 2 computes

\[ S_3 = \text{Goto}(S_1, \text{ baa}) = \{ [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \cdot, \text{EOF}], [\text{SheepNoise} \rightarrow \text{SheepNoise} \text{ baa} \cdot, \text{ baa}] \} \]
The algorithm produces the following table

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>baa</td>
<td>eof</td>
</tr>
<tr>
<td></td>
<td>SheepNoise</td>
</tr>
<tr>
<td>0</td>
<td>s 2</td>
</tr>
<tr>
<td>1</td>
<td>s 3</td>
</tr>
<tr>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>2</td>
<td>r 3</td>
</tr>
<tr>
<td>3</td>
<td>r 2</td>
</tr>
</tbody>
</table>
The algorithm produces the following table

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>baa</td>
<td>eof</td>
</tr>
<tr>
<td>0</td>
<td>s 2</td>
</tr>
<tr>
<td>1</td>
<td>s 3</td>
</tr>
<tr>
<td>2</td>
<td>r 3</td>
</tr>
<tr>
<td>3</td>
<td>r 2</td>
</tr>
<tr>
<td></td>
<td>SheepNoise</td>
</tr>
</tbody>
</table>

How to parse input: baa baa baa eof
More Syntax Analysis (bottom-up)

Error correction

Read EaC: Chapter 3.4