Bottom-Up Parsing II

CS143
Lecture 8

Instructor: Fredrik Kjolstad
Slide design by Prof. Alex Aiken, with modifications
Review: Shift-Reduce Parsing

Bottom-up parsing uses two actions:

Shift

\[
ABC \mid xyz \Rightarrow ABCx \mid yz
\]

Reduce

\[
Cbxy \mid ijk \Rightarrow CbA \mid ijk
\]
Recall: The Stack

• Left string can be implemented by a stack
  – Top of the stack is the $|$,

• Shift pushes a terminal on the stack

• Reduce
  – pops 0 or more symbols off of the stack
    • production rhs
  – pushes a non-terminal on the stack
    • production lhs
Key Issue

• How do we decide when to shift or reduce?

• Example grammar:

\[
E \rightarrow T + E \mid T \\
T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
\]

• Consider step \text{int} \mid * \text{int} + \text{int}
  – We could reduce by \text{T} \rightarrow \text{int} giving \text{T} \mid * \text{int} + \text{int}
  – A fatal mistake!
    • No way to reduce to the start symbol \text{E}
Definition: Handles

• Intuition: Want to reduce only if the result can still be reduced to the start symbol

• Assume a rightmost derivation

\[ S \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega \]

• Then \( X \rightarrow \beta \) in the position after \( \alpha \) is a handle of \( \alpha \beta \omega \)

• Can and must reduce at handles
• Handles formalize the intuition
  – A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

• We only want to reduce at handles

• Note: We have said what a handle is, not how to find handles
Important Fact #2

Important Fact #2 about bottom-up parsing:

In shift-reduce parsing, handles appear only at the top of the stack, never inside
Why?

• Informal induction on # of reduce moves:

• True initially, stack is empty

• Immediately after reducing a handle
  – right-most non-terminal on top of the stack
  – next handle must be to right of right-most non-terminal, because this is a right-most derivation
  – Sequence of shift moves reaches next handle
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack.

- Handles are never to the left of the rightmost non-terminal.
  - Therefore, shift-reduce moves are sufficient; the parser need never move left.

- Bottom-up parsing algorithms are based on recognizing handles.
Recognizing Handles

- There are no known efficient algorithms to recognize handles
- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars
Grammars

All CFGs

Unambiguous CFGs

SLR CFGs

LR(0) CFGs

will generate conflicts
Viable Prefixes

• It is not obvious how to detect handles

• At each step the parser sees only the stack, not the entire input; start with that . . .

\[ \alpha \] is a viable prefix if there is an \( \omega \) such that \( \alpha | \omega \) is a state of a shift-reduce parser
Huh?

- What does this mean? A few things:
  - A viable prefix does not extend past the right end of the handle
  - It’s a viable prefix because it is a prefix of the handle
  - As long as a parser has viable prefixes on the stack no parsing error has been detected
Important Fact #3

Important Fact #3 about bottom-up parsing:

For any SLR grammar, the set of viable prefixes is a regular language
Important Fact #3 (Cont.)

- Important Fact #3 is non-obvious
- We show how to compute automata that accept viable prefixes
Items

- An item is a production with a “.” somewhere on the rhs, denoting a focus point

- The items for $T \rightarrow (E)$ are
  - $T \rightarrow .(E)$
  - $T \rightarrow (.E)$
  - $T \rightarrow (E.)$
  - $T \rightarrow (E).$
Items (Cont.)

- The only item for $X \rightarrow \varepsilon$ is $X \rightarrow \cdot$.

- Items are often called “LR(0) items”
Intuition

• The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions
  – If it had a complete rhs, we could reduce

• These bits and pieces are always prefixes of rhs of productions
Example

Consider the input (int)

- Then \((E | \) is a state of a shift-reduce parse

- \((E \) is a prefix of the rhs of \( T \rightarrow (E) \)
  - Will be reduced after the next shift

- Item \( T \rightarrow (E.) \) says that so far we have seen \((E \) of this production and hope to see \)
Generalization

• The stack may have many prefixes of rhs’s
  \( \text{Prefix}_1 \text{ Prefix}_2 \ldots \text{Prefix}_{n-1} \text{ Prefix}_n \)

• Let \( \text{Prefix}_i \) be a prefix of rhs of \( X_i \rightarrow \alpha_i \)
  – \( \text{Prefix}_i \) will eventually reduce to \( X_i \)
  – The missing part of \( \text{Prefix}_{i-1} \) of \( \alpha_{i-1} \) starts with \( X_i \)
  – i.e. there is a \( X_{i-1} \rightarrow \text{Prefix}_{i-1} \; X_i \beta \) for some \( \beta \)

• Recursively, \( \text{Prefix}_{k+1} \ldots \text{Prefix}_n \) eventually reduces to the missing part of \( \alpha_k \)
An Example

Consider the string \((\text{int} \* \text{int})\):

\((\text{int} \* \text{int})\) is a state of a shift-reduce parse

From top of the stack:

“\(\varepsilon\)” is a prefix of the rhs of \(E \rightarrow T\)

“(“ is a prefix of the rhs of \(T \rightarrow (E)\)

“\(\varepsilon\)” is a prefix of the rhs of \(E \rightarrow T\)

“\(\text{int} \*\)” is a prefix of the rhs of \(T \rightarrow \text{int} \* T\)
The stack of items

\[
\begin{align*}
T & \rightarrow \text{int * } .T \\
E & \rightarrow .T \\
T & \rightarrow (.E)
\end{align*}
\]

Says

We’ve seen \text{int *} of \(T \rightarrow \text{int * } T\)

We’ve seen \(\varepsilon\) of \(E \rightarrow T\)

We’ve seen \((\) of \(T \rightarrow (E)\)
Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs’s of productions, where

- Each sequence can eventually reduce to part of the missing suffix of its predecessor
An NFA Recognizing Viable Prefixes

1. Add a new start production $S' \rightarrow S$ to $G$

2. The NFA states are the items of $G$
   - (Including the new start production)

3. For item $E \rightarrow \alpha.X\beta$ add transition
   $$E \rightarrow \alpha.X\beta \xrightarrow{X} E \rightarrow \alpha X.\beta$$

4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
   $$E \rightarrow \alpha.X\beta \xrightarrow{\varepsilon} X \rightarrow \cdot \gamma$$
An NFA Recognizing Viable Prefixes (Cont.)

5. Every state is an accepting state

6. Start state is $S' \rightarrow .S$
NFA for Viable Prefixes

S' → . E

E → T + E | T
T → int * T | int | (E)
NFA for Viable Prefixes

E → T + E | T
T → int * T | int | (E)

\[ S' \rightarrow E. \]

\[ S' \rightarrow .E \]

\[ E \rightarrow .T+E \]

\[ E \rightarrow .T \]

\[ \epsilon \]
NFA for Viable Prefixes

E → T + E | T
T → int * T | int | (E)

E → T.
T → . (E)
S’ → E.
S’ → . E
E → . T
E → . T+E
T → . int
T → . int * T
E → T.
NFA for Viable Prefixes

E \rightarrow T + E \mid T
T \rightarrow \text{int} * T \mid \text{int} \mid (E)

\[
\begin{array}{c}
T \rightarrow .(E) \\
S' \rightarrow E. \\
S' \rightarrow .E \\
E \rightarrow .T \\
E \rightarrow .T+E \\
E \rightarrow T.+E \\
E \rightarrow T. \\
T \rightarrow .\text{int} \\
T \rightarrow .\text{int} * T \\
E \rightarrow T. \\
\end{array}
\]
NFA for Viable Prefixes

\[
\begin{align*}
T & \rightarrow \cdot (E) \\
S' & \rightarrow E. \\
S' & \rightarrow \cdot E \\
E & \rightarrow \cdot T+E \\
E & \rightarrow \cdot T \\
E & \rightarrow T+ E | T \ | \int \ | (E) \\
T & \rightarrow \cdot \int * T \\
T & \rightarrow \cdot \int \\
E & \rightarrow T. \\
E & \rightarrow T+ E \\
S' & \rightarrow E. \\
S' & \rightarrow \cdot E \\
E & \rightarrow \cdot T+E \\
E & \rightarrow \cdot T \\
E & \rightarrow T+ E | T \ | \int \ | (E) \\
T & \rightarrow \cdot \int * T \\
T & \rightarrow \cdot \int \\
E & \rightarrow T. \\
\end{align*}
\]
NFA for Viable Prefixes

\[ E \rightarrow T + E \mid T \]
\[ T \rightarrow \text{int} \ast T \mid \text{int} \mid (E) \]
NFA for Viable Prefixes

\[ S' \rightarrow E. \]
\[ S' \rightarrow .E \]
\[ E \rightarrow .T+E \]
\[ E \rightarrow T+E | T \]
\[ T \rightarrow .int \]
\[ T \rightarrow .int * T \]
\[ T \rightarrow (E.) \]
\[ T \rightarrow (E) \]
\[ E \rightarrow T + E | T | \int | (E) \]
NFA for Viable Prefixes

E → T + E | T
T → int * T | int | (E)
NFA for Viable Prefixes

E → T + E | T
T → int * T | int | (E)

T → . (E)

T → (.E)

T → (E.)

T → (E).

S' → E.

S' → . E

E → . T + E

E → T + E

E → T + . E

E → T + . E.

E → T + E.

E → T.

E → T.

E → T.

T → . int

T → . int

T → . int * T

int → T → int.

+ → E → T + . E

ε → T → . int

ε → T → . int

ε → T → . int

ε → T → . int

ε → T → . int
NFA for Viable Prefixes

E → T + E | T
T → int * T | int | (E)

E → .(E)
T → (.E)
T → (E.)
T → (E).
T → int
T → int. * T
T → int.
T → int. * T
E → T + E
E → T + E.
E → T + .E
E → T. + E
E → . T + E
E → . T + E.
E → . T
E → T.
S’ → E.
E → T
E → T.
S’ → E
E → T
E → T.
E → T.
E → T.
E → T.
E → T.
NFA for Viable Prefixes

\[ E \rightarrow T \mid E \mid T \]

\[ T \rightarrow \text{int} \]

\[ T \rightarrow \text{int} \ast T \]

\[ T \rightarrow (E.) \]

\[ E \rightarrow .(E) \]

\[ S' \rightarrow E. \]

\[ S' \rightarrow .E \]

\[ E \rightarrow .T \]

\[ E \rightarrow .T + E \]

\[ E \rightarrow T + .E \]

\[ E \rightarrow T + .E. \]

\[ T \rightarrow \text{int} \ast .T \]

\[ T \rightarrow \text{int} \ast T \]
NFA for Viable Prefixes

E → T + E | T
T → int * T | int | (E)
Translation to the DFA

S' → E .
E → T .
E → T . + E
T → .(E)
T → .int * T
T → .int

E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

E → T + E .
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

T → (E .)
T → (E .)
T → .int
The states of the DFA are 

“canonical collections of items”

or

“canonical collections of LR(0) items”

The Dragon book gives another way of constructing LR(0) items
Valid Items

Item $X \rightarrow \beta.\gamma$ is valid for a viable prefix $\alpha\beta$ if

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\gamma\omega$$

by a right-most derivation

After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items
An item $I$ is valid for a viable prefix $\alpha$ if the DFA recognizing viable prefixes terminates on input $\alpha$ in a state $s$ containing $I$.

The items in $s$ describe what the top of the item stack might be after reading input $\alpha$. 
Valid Items Example

• An item is often valid for many prefixes

• Example: The item $T \rightarrow (.E)$ is valid for prefixes

  ( 
    ( 
      ( 
        ( 
          ( 
            ( . . . 
          ) 
        ) 
      ) 
    ) 
  ) 
  . . .
LR(0) Parsing

• Idea: Assume
  – stack contains \( \alpha \)
  – next input is \( t \)
  – DFA on input \( \alpha \) terminates in state \( s \)

• Reduce by \( X \rightarrow \beta \) if
  – \( s \) contains item \( X \rightarrow \beta \).

• Shift if
  – \( s \) contains item \( X \rightarrow \beta . t \omega \)
  – equivalent to saying \( s \) has a transition labeled \( t \)
LR(0) Conflicts

• LR(0) has a reduce/reduce conflict if:
  – Any state has two reduce items:
    – $X \rightarrow \beta$. and $Y \rightarrow \omega$.

• LR(0) has a shift/reduce conflict if:
  – Any state has a reduce item and a shift item:
    – $X \rightarrow \beta$. and $Y \rightarrow \omega.t\delta$
Translation to the DFA

Two shift/reduce conflicts with LR(0) rules
SLR

- LR = “Left-to-right scan”
- SLR = “Simple LR”

- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts
SLR Parsing

- Idea: Assume
  - stack contains $\alpha$
  - next input is $t$
  - DFA on input $\alpha$ terminates in state $s$

- Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$.
  - $t \in \text{Follow}(X)$

- Shift if
  - $s$ contains item $X \rightarrow \beta.t\omega$
SLR Parsing (Cont.)

• If there are conflicts under these rules, the grammar is not SLR

• The rules amount to a heuristic for detecting handles
  – The SLR grammars are those where the heuristics detect exactly the handles
Translation to the DFA

Follow(E) = { ‘)’, $ } 
Follow(T) = { ‘+’, ‘)’, $ }

No conflicts with SLR rules!
Precedence Declarations Digression

• Lots of grammars aren’t SLR
  – including all ambiguous grammars

• We can parse more grammars by using precedence declarations
  – Instructions for resolving conflicts
Consider our favorite ambiguous grammar:

\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid \text{int} \]

The DFA for this grammar contains a state with the following items:

\[ E \rightarrow E \ast E \mid E \rightarrow E \ast + E \]

- shift/reduce conflict!

Declaring “\* has higher precedence than +” resolves this conflict in favor of reducing.
Precedence Declarations (Cont.)

• The term “precedence declaration” is misleading

• These declarations do not define precedence; they define conflict resolutions
  – Not quite the same thing!
Naïve SLR Parsing Algorithm

1. Let $\mathbf{M}$ be DFA for viable prefixes of $\mathbf{G}$
2. Let $|x_1...x_n|$ be initial configuration
3. Repeat until configuration is $S\$$
   • Let $\alpha \omega$ be current configuration
   • Run $\mathbf{M}$ on current stack $\alpha$
   • If $\mathbf{M}$ rejects $\alpha$, report parsing error
     • Stack $\alpha$ is not a viable prefix
   • If $\mathbf{M}$ accepts $\alpha$ with items $\mathbf{I}$, let $t$ be next input
     • Reduce if $X \rightarrow \beta. \in \mathbf{I}$ and $t \in \text{Follow}(X)$
     • Otherwise, shift if $X \rightarrow \beta. t \gamma \in \mathbf{I}$
     • Report parsing error if neither applies
Notes

• If there is a conflict in the last step, grammar is not SLR(k)

• $k$ is the amount of lookahead
  – In practice $k = 1$

• Will skip using extra start state $S'$ in following example to save space on slides
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int * int$</code></td>
<td>1</td>
<td><code>shift</code></td>
</tr>
</tbody>
</table>
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l \text{ int } * \text{ int}$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int $l \text{ int}$</td>
<td>3</td>
<td>* not in Follow($T$) shift</td>
</tr>
</tbody>
</table>

E → T + E | T → int * T | l int l (E)
int $\ast$ int$^*$
**SLR Example**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$</td>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$</td>
<td>11</td>
<td>shift</td>
</tr>
</tbody>
</table>

**Grammar**

\[
E \rightarrow T + E \mid T \\
T \rightarrow \text{int} * T \mid \text{int} \mid \text{(E)}
\]
int * | int$

S' \rightarrow E.
E \rightarrow . T
E \rightarrow . T + E
T \rightarrow .(E)
T \rightarrow .int * T
T \rightarrow .int

E \rightarrow T + . E
E \rightarrow .T
E \rightarrow .T + E
T \rightarrow .(E)
T \rightarrow .int * T
T \rightarrow .int

T \rightarrow int * .T
T \rightarrow .(E)
T \rightarrow .int * T
T \rightarrow .int

T \rightarrow (E.)
T \rightarrow (E.
T \rightarrow .(E)
T \rightarrow .int * T
T \rightarrow .int

E \rightarrow T + E
E \rightarrow .T
E \rightarrow .T + E
T \rightarrow .(E)
T \rightarrow .int * T
T \rightarrow .int
int * | int$
int * l int$
## SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$</td>
<td>3</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$</td>
<td>11</td>
<td>shift</td>
</tr>
<tr>
<td>int * int</td>
<td>3</td>
<td>$ ∈ Follow(T)</td>
</tr>
</tbody>
</table>

**Grammar:**

\[
E \rightarrow T + E | T | \text{(E)} \\
T \rightarrow \text{int} * T | \text{int} | (E)
\]
int * int \mid $
int * int $
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l \text{ int } * \text{ int}$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>$\text{int } l * \text{ int}$</td>
<td>3 * not in Follow(T)</td>
<td>shift</td>
</tr>
<tr>
<td>$\text{int } * \text{ int}$</td>
<td>11</td>
<td>shift</td>
</tr>
<tr>
<td>$\text{int } * \text{ int } l$</td>
<td>3 $ \in $ Follow(T)</td>
<td>reduce $T \rightarrow \text{int}$</td>
</tr>
<tr>
<td>$\text{int } * \text{T } l$</td>
<td>4 $ \in $ Follow(T)</td>
<td>reduce $T \rightarrow \text{int*T}$</td>
</tr>
</tbody>
</table>
int * T | $
# SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>I int * int$</code></td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td><code>int I * int$</code></td>
<td>3 * not in Follow(T)</td>
<td>shift</td>
</tr>
<tr>
<td><code>int * I int$</code></td>
<td>11</td>
<td>shift</td>
</tr>
<tr>
<td><code>int * int I$</code></td>
<td>3 $ ∈ Follow(T)</td>
<td>reduce T → int</td>
</tr>
<tr>
<td><code>int * T I$</code></td>
<td>4 $ ∈ Follow(T)</td>
<td>reduce T → int*T</td>
</tr>
<tr>
<td><code>T I$</code></td>
<td>5 $ ∈ Follow(T)</td>
<td>reduce E → T</td>
</tr>
</tbody>
</table>
## SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>l int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int l * int$</td>
<td>3 * not in Follow(T)</td>
<td>shift</td>
</tr>
<tr>
<td>int * l int$</td>
<td>11</td>
<td>shift</td>
</tr>
<tr>
<td>int * int l$</td>
<td>3 $ \in$ Follow(T)</td>
<td>reduce T$\rightarrow$int</td>
</tr>
<tr>
<td>int * T l$</td>
<td>4 $\in$ Follow(T)</td>
<td>reduce T$\rightarrow$int*T</td>
</tr>
<tr>
<td>T l$</td>
<td>5 $\in$ Follow(T)</td>
<td>reduce E$\rightarrow$T</td>
</tr>
<tr>
<td>E l$</td>
<td></td>
<td>accept</td>
</tr>
</tbody>
</table>
An Improvement

• Rerunning the automaton at each step is wasteful
  – Most of the work is repeated

• Remember the state of the automaton on each prefix of the stack

• Change stack to contain pairs
  \( \langle \text{symbol}, \text{DFA state} \rangle \)
An Improvement (Cont.)

• For a stack

\[ \langle \text{symbol}_1, \text{state}_1 \rangle \ldots \langle \text{symbol}_n, \text{state}_n \rangle \]

\text{state}_n \text{ is the final state of the DFA on symbol}_1\ldots\text{symbol}_n

• Detail: The bottom of the stack is \( \langle \text{dummy}, \text{start} \rangle \)
where
  – \text{dummy} is a dummy symbol
  – \text{start} is the start state of the DFA
Goto (DFA) Table

- Define $\text{goto}[i,A] = j$ if $\text{state}_i \rightarrow^A \text{state}_j$

- $\text{goto}$ is just the transition function of the DFA
  - One of two parsing tables
Refined Parser Moves

• **Shift x**
  – Push \( \langle a, x \rangle \) on the stack
  – \( a \) is current input
  – \( x \) is a DFA state

• **Reduce** \( X \rightarrow \alpha \)
  – As before

• **Accept**

• **Error**
Action Table

For each state $s_i$ and terminal $t$

- If $s_i$ has item $X \rightarrow \alpha.t\beta$ and $\text{goto}[i,t] = k$
  then $\text{action}[i,t] = \text{shift } k$

- If $s_i$ has item $X \rightarrow \alpha.$ and $t \in \text{Follow}(X)$ and $X \neq S'$ then
  $\text{action}[i,t] = \text{reduce } X \rightarrow \alpha$

- If $s_i$ has item $S' \rightarrow S.$ then $\text{action}[i,\$] = \text{accept}$

- Otherwise, $\text{action}[i,t] = \text{error}$
SLR Parsing Algorithm

Let input = w$ be initial input
Let j = 0
Let DFA state 1 be the one with item S’ → .S
Let stack = 〈 dummy, 1 〉 // 〈 symbol, state 〉

repeat
  case action[top_state(stack), input[j]] of
    shift k: push 〈 input[j++], k 〉
    reduce X → α:
      pop |α| pairs,
      push 〈 X, goto[top_state(stack), X] 〉
    accept: halt normally
    error: halt and report error
Notes on SLR Parsing Algorithm

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!

- However, we still need the symbols for semantic actions
More Notes

• Some common constructs are not SLR(1)

• LR(1) is more powerful
  – Build lookahead into the items
  – An LR(1) item is a pair: (LR(0) item, x lookahead)
  – \([T \rightarrow . \text{int} * T, \$]\) means
    • After seeing \(T \rightarrow \text{int} * T\) reduce if lookahead is \$
  – More accurate than just using follow sets
  – See Dragon Book
  – Take a look at the LR(1) automaton for your parser!