Status

- Project 1 Back By Friday
- Project 2 Due Friday, Apr. 24, 11:55pm
- My office hours 10am today
Parsers

- Parsers are built from CFGs
- CFGs consist of *terminals* (tokens) and non-*terminals* (rules or production)
- A *production* has a left hand side and a right hand side, separated by an arrow, e.g., $X \rightarrow \alpha\beta$
- Any language that can be described by an RE can be described by a CF language. The opposite is not true.
Derivation Process

1. Start with $S$, which is the starting non-terminal.

2. Let $S = \alpha$

3. If $\alpha$ is a terminal (incl. empty), stop. Alpha is a string in the CFG.

4. If $\alpha$ is not a terminal, choose a rule with $\alpha$ on the LHS) and replace LHS with RHS.

5. Continue (4) until $X \rightarrow \alpha \cdot \beta$, $a$

6. $\alpha$ is eventually a string in the CFG. (So is $S$).
\begin{align*}
E & \rightarrow E + E & \text{Derive a string: } -8 * A \\
E & \rightarrow E * E \\
E & \rightarrow -E \\
E & \rightarrow \text{id} \\
E & \rightarrow \text{num} \\
E & \rightarrow (E)
\end{align*}

\begin{align*}
E & \quad \text{start state} \\
E * E & \quad \text{pick rule, replace LHS with RHS} \\
-E*E & \quad \text{again, pick rule, replace LHS with RHS} \\
-8*E & \quad \text{again, pick rule, replace LHS with RHS} \\
-8*A & \quad \text{done}
\end{align*}
- root is the starting non-terminal
- nodes are non-terminals
- leaves are terminals
- reads in L to R order (depth first) to give string
Ambiguity

• Def.: left-most derivation — during each step (in top-down parsing), substitute the left-most non-terminals

• Def.: a CFG is ambiguous if for some CFG w/string x, there exist:
  • 2 different left-most derivations, or
  • 2 different parse trees for x
Ex.:
\[ E \rightarrow E + E \]
\[ E \rightarrow E * E \]
\[ E \rightarrow \text{num} \]

Two different, valid parse trees

\[ 2 + 3 * 8 \]
Two Left-Most Derivations

E
E + E
2 + E
2 + E * E
2 + 3 * E
2 + 3 * 8

E
E * E
E + E * E
2 + E * E
2 + 3 * E
2 + 3 * 8

grammar is ambiguous
How do you fix an ambiguous grammar?

- Determine what the language is
- Find the bad string(s)
- For each bad string change the grammar to disallow the derivation
Example

What kind of string is this describing?

\[
\begin{align*}
S & \rightarrow X \\
S & \rightarrow YZ \\
X & \rightarrow aXb \\
X & \rightarrow \varepsilon \\
Y & \rightarrow abY \\
Y & \rightarrow ab \\
Z & \rightarrow a \\
Z & \rightarrow \varepsilon
\end{align*}
\]
Example

What kind of string is this describing?

\[ S \rightarrow X \]
\[ S \rightarrow YZ \]
\[ X \rightarrow aXb \]
\[ X \rightarrow \varepsilon \]
\[ Y \rightarrow abY \]
\[ Y \rightarrow ab \]
\[ Z \rightarrow a \]
\[ Z \rightarrow \varepsilon \]

\[ X = a^n b^n \text{ where } n \geq 0 \]

\[ YZ = (ab)^+ a? \]

What is an example of a bad string?
ba — two parse trees
Need to Change Grammar

\[
\begin{align*}
  S & \to X \\
  S & \to YZ \\
  X & \to aXb \\
  X & \to \varepsilon \\
  Y & \to abY \\
  Y & \to ab \\
  Z & \to a \\
  Z & \to \varepsilon
\end{align*}
\]
Incorrect Disambiguation

- Can’t change grammar so that bad strings are now allowed
- Can’t change grammar so that some good strings are now disallowed
Option 1

Change $X = a^n b^n$ where $n \geq 0$
to $X = a^n b^n$ where $n = 0$ or $n \geq 2$ (let YZ take $n = 1$)

Change this:

$X \rightarrow aXb$
$X \rightarrow \varepsilon$

To this:

$X' \rightarrow aabb$  // here’s $n = 2$
$X' \rightarrow aX'b$  // here’s $n > 2$
$X \rightarrow X'$
$X \rightarrow \varepsilon$  // here’s $n = 0$
Option 2

Change $Y = (ab)^+$ to $Y = ab(ab)^+$ so let $X$ take the $n = 1$ case

Change this:

$Y \rightarrow abY$
$Y \rightarrow ab$

To this:

$Y \rightarrow abY$
$Y \rightarrow abab$
Full Grammar With Option 2

\[
\begin{align*}
S & \to X \\
S & \to YZ \quad \text{No longer has “ab” ambiguity} \\
X & \to aXb \quad \text{No longer allows } S \to YZ \to abZ \to aba \\
X & \to \varepsilon \\
Y & \to abY \\
Y & \to abab \quad (\text{used to be } Y \to ab) \\
Z & \to a \\
Z & \to \varepsilon \quad \text{So it is incorrect!}
\end{align*}
\]
Fixing Grammars

• No algorithm for fixing grammars.

• Fixing grammars requires intuition, experimentation and many errors (very frustrating).
Bottom-Up Parsing
Lecture 8
Bottom-Up Parsing

- Also called “LR parsing”
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation
Idea

• LR parsing *reduces* a string to the start symbol by inverting productions

• str ← input string of terminals

• repeat
  • Identify β in str such that A → β is a production (i.e., str = α β γ)
  • Replace β by A in str (i.e., str becomes α A γ)
  • until str = S
Important Fact #1

• An LR parser traces a rightmost derivation in reverse.
Shift

- \textit{Shift}: Move $\triangleright$ one place to the right
- Shifts a terminal to the left string

\[ E + (\triangleright \text{int}) \Rightarrow E + (\text{int} \triangleright) \]
Reduce

- **Reduce**: Apply an inverse production at the right end of the left string

If $E \rightarrow E + (E)$ is a production, then:

$$E + (E + (E) \quad \Rightarrow \quad E + (E \quad \Rightarrow)$$
The Stack

• Left string can be implemented as 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) and pushes a non-termianl on the stack (production lhs)
When to Shift vs. Reduce?

- Decide based on the left string (the stack)
- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The DFA input is the stack
  - The language consists of terminals and non-terminals
- We run the DFA on the stack and we examine the resulting state \( \mathsf{X} \) and the token \( \mathsf{T} \) after △
  - If \( \mathsf{X} \) has a transition labeled \( \mathsf{T} \) then \textit{shift}
  - If \( \mathsf{X} \) is labeled with “\( \mathsf{A} \rightarrow \mathsf{B} \) on \( \mathsf{T} \)” then \textit{reduce}
Representing the DFA

- Parsers represent the DFA as a 2D table.
- Recall table-driven lexing
- Rows correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table
## Example Decision Table

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>c</th>
<th>i</th>
<th>m</th>
<th>r</th>
<th>$</th>
<th>S</th>
<th>D</th>
<th>L</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How is the DFA Constructed?

- The stack describes the “context” of the parse
  - What non-terminal we are looking for
  - What production rhs we are looking for
  - What we have seen so far from the rhs
- Each DFA state describes several such contexts
  - E.g., when we are looking for non-terminal E, we might be looking either for an int or an E+(E) rhs
LR(1) Items

- An **LR(1) item** is a pair, e.g.,:
  \[ X \rightarrow \alpha \cdot \beta, a \]
- \( X \rightarrow \alpha \beta \) is a production
- \( a \) is a terminal (the lookahead terminal)
- LR(1) means 1 lookahead terminal
- \( [X \rightarrow \alpha \cdot \beta, a] \) describes a context of the parser
  - We are trying to find an \( X \) followed by an \( a \), and
  - We already have \( \alpha \) on the top of the stack
  - Thus we need to see a prefix derived from \( \beta a \)
Note

• Symbol “•” separates stack from rest of input: \( \alpha \cdot \gamma \)
  where \( \alpha \) is the stack and \( \gamma \) is the remaining string of terminals.

• In LR(1) items, • is used to mark a prefix of a production RHS:
  \[ X \rightarrow \alpha \cdot \beta, a \]
  • Here, \( \beta \) might contain non-terminals as well.

• In both cases, the stack is on the left.
LR(1) Items

- In context containing
  \[ E \rightarrow E + \cdot (E), + \]
- If ‘(‘ follows then we can perform a shift to context containing
  \[ E \rightarrow E + (\cdot E), + \]
- In context containing
  \[ E \rightarrow E + (E)\cdot, + \]
- We can perform a reduction with \[ E \rightarrow E + (E) \]
- But only if a + follows
LR(1) Items

• Consider a context with the item
  \[ E \rightarrow E + (\cdot E), + \]
• We expect a string derived from \[ E \) +
• There are two productions for \( E \)
  \[ E \rightarrow E + (E) \quad \text{and} \quad E \rightarrow \text{int} \]
• We describe this by extending the context with two more items:
  \[ E \rightarrow \cdot \text{int}, ) \]
  \[ E \rightarrow \cdot E + (E), ) \]
First and Follows

- Consider the state $S \rightarrow \beta \cdot A \gamma$
- We’re trying to match the string $\beta b \delta$
- Suppose that $b$ is the next token.
- Either:
  - $A \rightarrow \alpha$, if $b$ can start a string derived from $\alpha$
    - We say that $b \in \text{First}(\alpha)$
  - Or, the expansion of $A$ is empty and $b$ belongs to an expansion of $\gamma$ (e.g., $\gamma \rightarrow b \omega$).
  - $b$ can appear after $A$ in a derivation of the form $S \rightarrow \beta A b \omega$
    - We say that $b \in \text{Follow}(A)$ in this case.
What productions can we use?

- Consider the state $S \rightarrow \beta \cdot A\gamma$
- We’re trying to match the string $\beta b\delta$
  - The expansion of $A$ is empty and $b$ belongs to an expansion of $\gamma$ (e.g., $\gamma \rightarrow b\omega$).
  - $b$ can appear after $A$ in a derivation of the form $S \rightarrow \cdot \beta Ab\omega$
  - We say that $b \in \text{Follow}(A)$ in this case.

- **Can use as a production:**
  - $A \rightarrow \alpha$ can be used if $\alpha$ can expand to $\varepsilon$
  - We say that $\varepsilon \rightarrow \text{First}(A)$ in this case
Computing First Sets

- Definition: First(X) = \{ b | X \rightarrow b\alpha \} \cup \{ \varepsilon | X \rightarrow \varepsilon \}
- First (b) = \{ b \}
- For all productions X \rightarrow A_1 ... A_n
  - Add First (A_1) - \{ \varepsilon \} to First(X). Stop if \varepsilon \notin First(A_1)
  - ...  
  - Add First (A_n) - \{ \varepsilon \} to First(X). Stop if \varepsilon \notin First(A_n)
  - Add \varepsilon to First(X) (ignore A_i if it is X)
First sets Example

• Grammar:
  \[ E \rightarrow TX \]
  \[ T \rightarrow (E) \mid \text{int } Y \]
  \[ X \rightarrow + E \mid \varepsilon \]
  \[ Y \rightarrow *T \mid \varepsilon \]

• First sets
  \[ \text{First } (() ) = \{ ( ) \} \]
  \[ \text{First } ( ) ) = \{ ) \} \}
  \[ \text{First } ( \text{int } ) = \{ \text{int } \} \]
  \[ \text{First } ( + ) = \{ + \} \]
  \[ \text{First } ( * ) = \{ * \} \]
  \[ \text{First } ( T ) = \{ \text{int }, ( ) \} \]
  \[ \text{First } ( E ) = \{ \text{int }, ( ) \} \]
  \[ \text{First } ( X ) = \{ +, \varepsilon \} \]
  \[ \text{First } ( Y ) = \{ *, \varepsilon \} \]
Computing *Follow* Sets

- Definition: \( \text{Follow}(X) = \{ b \mid S \rightarrow \beta X b \omega \} \)
- Compute the First sets for all non-terminals first
- Add $ to \text{Follow}(S)$ (if S is the start non-terminal)
- For all productions \( Y \rightarrow X A_1 \ldots A_n \)
  - Add First (\( A_1 \)) - \{\( \varepsilon \)\} to \text{Follow}(X). Stop if \( \varepsilon \not\in \text{First}(A_1) \)
  - ...
  - Add First (\( A_n \)) - \{\( \varepsilon \)\} to \text{Follow}(X). Stop if \( \varepsilon \not\in \text{First}(A_n) \)
- Add \text{Follow}(Y) to \text{Follow}(X)
Follow sets Example

• Grammar:

\[
E \rightarrow TX \\
T \rightarrow (E) \mid \text{int } Y \\
X \rightarrow + E \mid \varepsilon \\
Y \rightarrow *T \mid \varepsilon
\]

• First sets

Follow ( + ) = \{ \text{int, ( } ) \} \\
Follow ( ( ) ) = \{ \text{int, ( } ) \} \\
Follow ( X ) = \{ \$, ) \} \\
Follow ( ) ) = \{ +, ), $ \} \\
Follow ( \text{int } ) = \{ *, +, ), $ \}

Follow ( * ) = \{ \text{int, ( } ) \} \\
Follow ( E ) = \{ ), $ \} \\
Follow ( T ) = \{ +, ), $ \} \\
Follow ( Y ) = \{ +, ), $ \}
LR(1) Items

- Consider a context with the item
  \[ E \rightarrow E + (\cdot E), + \]
- We expect a string derived from \( E \) \( ) \) \( + \)
- There are two productions for \( E \)
  \[ E \rightarrow E + (E) \quad \text{and} \quad E \rightarrow \text{int} \]
- We describe this by extending the context with two more items:
  \[ E \rightarrow \cdot \text{int}, ) \]
  \[ E \rightarrow \cdot E + (E) , ) \]
The Closure Operation

• The operation of extending the context with items is called the closure operation

\[
\text{Closure}(\text{Items}) = \\
\text{repeat} \\
\text{for each } [X \rightarrow \alpha \cdot \beta, a] \text{ in Items} \\
\text{for each production } Y \rightarrow \gamma \\
\text{for each } b \in \text{First}(\beta a) \\
\text{add } [Y \rightarrow \cdot \gamma, b] \text{ to Items} \\
\text{until Items is unchanged}
\]
Construct the Parsing DFA

• Construct the start context: \( \text{Closure} \{ S \rightarrow \cdot E, \$ \} \)

\[
\begin{align*}
S & \rightarrow \cdot E, \$ \\
E & \rightarrow \cdot E+(E), \$ \\
E & \rightarrow \cdot \text{int}, \$
\end{align*}
\]

• We abbreviate as:

\[
\begin{align*}
S & \rightarrow \cdot E, \$
E & \rightarrow \cdot E+(E), $/+
E & \rightarrow \cdot \text{int}, $, $/+ \\
E & \rightarrow \cdot \text{int}, +
\end{align*}
\]
Construct the Parsing DFA

• A DFA state is a *closed* set of LR(1) items

• This means that we performed Closure

• The start state contains \([S \rightarrow \cdot E, $]\)

• A state that contains \([X \rightarrow \alpha \cdot, b]\) is labeled with “reduce with \(X \rightarrow \alpha\) on \(b\)”

• And now the transitions...
DFA Transitions

• A state “State” that contains $[X \rightarrow \alpha \cdot \gamma \beta, b]$ has a transition labeled $y$ to a state that contains the items “Transition(State, $\gamma$)”
  • $\gamma$ can be a terminal or a non-terminal

Transition(State, $\gamma$)

Items $\leftarrow \emptyset$

for each $[X \rightarrow \alpha \cdot \gamma \beta, b] \in \text{State}$

add $[X \rightarrow \alpha \gamma \cdot \beta, b]$ to Items

return Closure(Items)
We have defined several functions to help us on the way:

- closure \((s)\) - where \(s\) is a state
- goto \((S, X)\) where \(S\) is a set of LR items
- First \((X)\)
- Follows \((X)\)

...and for all cases \(X\) is a grammar symbol (either terminal or non-terminal)
Algorithms

- We have defined several algorithms which take us from the CFG to the decision table and the process which walks us through the parsing.
  - CFG $\rightarrow$ DFA
  - Fill in the action/goto entries in decision table
  - Using the DFA, a stack and action/goto tables, parse a grammar.
Example Grammar

S \rightarrow Xb
X \rightarrow YZ
Y \rightarrow aa
Z \rightarrow cS
Z \rightarrow \varepsilon
CFG $\rightarrow$ DFA

- Start by adding a new state to the grammar:

  \[ S' \rightarrow \varepsilon \cdot S \]

  Take the new state, add closure to get the new start state, which is state 1.
State 0: Closure of $S$

$S' \rightarrow \cdot S$

$S \rightarrow \cdot X b$
State 0: Closure of X

\[
S' \rightarrow \bullet S \\
S \rightarrow \bullet Xb \\
X \rightarrow \bullet YZ
\]
State 0: Closure of Y

\[ S' \rightarrow \cdot S \]
\[ S \rightarrow \cdot Xb \]
\[ X \rightarrow \cdot YZ \]
\[ Y \rightarrow \cdot aa \]
Loop through the states

- State 0
- mark it
- go through all the grammar symbols ($S, X, Y, Z, a, b, c$). If the state expects any of these symbols (i.e., the symbol precedes a “•”), add a transition.
Looping through the states: State 0

State 0:
S' $\rightarrow$ •S
S $\rightarrow$ •Xb
X $\rightarrow$ •YZ
Y $\rightarrow$ •aa

On “S” get:
state 1: S' $\rightarrow$ S•

On “X” get:
state 2: S $\rightarrow$ X•b

On “Y” get:
state 3: X $\rightarrow$ Y•Z

don’t forget closure!
Looping through the states: State 0

State 0:
S’ → •S
S → •Xb
X → •YZ
Y → •aa

On “S” get:
state 1: S’ → S•

On “X” get:
state 2: S → X•b

On “Y” get:
state 3: X → Y•Z
  Z → •cS
  Z → • ε
Looping through the states: State 0

State 0:
S' → •S
S → •Xb
X → •YZ
Y → •aa

On “a” get:
state 4: Y → a•a

On “b” or “c” get:
nothing (error)
Finishing the Decision Table

• Keep looping through all the other states, as you continue to produce them.

• What have you done? You’ve created the outline of a decision table with all of your states.

• But the decision table is empty.
Fill in actions/gotos

• Fill in shift actions (terminal transitions)
  • derive from the DFA. Look at the DFA. On state N, on input x, shift to state M.
  • action(N,x) = sM

• Fill in goto actions (terminal transitions)
  • derive from the DFA. In state R, on input Y, shift to state S
  • goto(R,Y) = S
Fill in accept state

- In state T, given this situation:
  
  $S' \rightarrow S$

  - We get action $(T,\$) = accept$
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>$</th>
<th>S</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Monday, April 27, 2009
What about reduce?

• An LR parser does a right-most derivation, which means the derivation is done in reverse.

• Given state $X \rightarrow abc$

• …the parser will recognize a reduce if the next input can "follow" $X$. 
Matching Patterns with Productions

• Back to the grammar, we add $ to $':

\[
\begin{align*}
S' & \rightarrow S $ \\
S & \rightarrow X \\
X & \rightarrow Y \\
Y & \rightarrow Z
\end{align*}
\]
Matching Patterns with Productions

- We start with $S' \rightarrow S$
- pattern: non terminal on end, add $\text{Follows}(S')$ to $\text{Follows}(S)$

$S' \ $ $
S \ $ $
X$
Y
Z
Matching Patterns with Productions

- We start with $S \rightarrow Xb$
- pattern: terminal following non terminal, add $b$ to $\text{Follows}(X)$

S'  $  
S   $  
X   b  
Y 
Z
Matching Patterns with Productions

- Looking at $X \rightarrow YZ$
- pattern: non-terminal (Z) at end, add $\text{Follows}(X)$ to $\text{Follows}(Z)$
- pattern: non-terminal (Y) followed by non-terminal (Z) going to empty, add $\text{Follows}(X)$ to $\text{Follows}(Y)$ (when Z is empty). Also, add $\text{First}(Z)$ to $\text{Follows}(Y)$, where $\text{First}(Z) = c$
Matching Patterns with Productions

\[ S' \rightarrow S \$
\[ S \rightarrow S \$
\[ X \rightarrow b
\[ Y \rightarrow b, c
\[ Z \rightarrow b \]
Okay, given this chart, we add actions. Look back to your states that you created from earlier. We only care about the ones which look like: $X \rightarrow \alpha^*$

- State 1 is $S' \rightarrow S\cdot$
- State 3 is $Z \rightarrow \cdot\varepsilon$
- State 5 is $S \rightarrow Xb\cdot$
- State 6 is $Z \rightarrow YZ\cdot$
- State 8 is $Y \rightarrow aa\cdot$
- State 9 is $Z \rightarrow cS\cdot$