CS536

Bottom-Up Parsing
Roadmap

• Last class
  – Name analysis

• Previous-ish last class
  – LL(1)

• Today’s class
  – LR Parsing
    • SLR(1)
Lecture Outline

• Introduce Bottom-Up parsing much like Top-Down
  – Talk about the language class / theory
  – Describe the state that it keeps / intuition
  – Show how it works
  – Show how it is built
LL(1) Not Powerful Enough for all PL

- Left-recursion
- Not left factored
- Doesn’t mean LL(1) is bad
  - Right tool for simple parsing jobs

```
stmtList ::= stmtList stmt
| /* epsilon */
;
```
We Need a *Little* More Power

- Could increase the lookahead
  - Up until the mid 90s, this was considered impractical
- Could increase the runtime complexity
  - CYK has us covered there
- Could increase the memory complexity
  - i.e. more elaborate parse table
LR Parsers

• Left-to-right scan of the input file
• Reverse rightmost derivation

• Advantages
  – Can recognize almost any programming language
  – Time and space $O(n)$ in the input size
  – More powerful than the corresponding LL parser i.e. LL(1) $< LR(1)$

• Disadvantages
  – More complex parser generation
  – Larger parse tables
LR Parser Power

• Let $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \omega$ be a rightmost derivation, where $\omega$ is a terminal string

• Let $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ be a step in the derivation
  • So $A \rightarrow \beta$ must have been a production in the grammar
  • $\alpha \beta \gamma$ must be some $\alpha_i$ or $\omega$

  – A grammar is LR(k) if for every derivation step, $A \rightarrow B$ can be inferred using only a scan of $\alpha \beta$ and at most $k$ symbols of $\gamma$

• Much like LL(1), you generally just have to go ahead and try it
LR Parser types

• LR(1)
  – Can recognize any DCFG
  – Can experience blowup in parse table size

• LALR(1)

• SLR(1)
  – Both proposed at the same time to limit parse table size
Which parser should we use?

• Different variants mostly differ in how they build the parse table, we can still talk about all the family in general terms
  – Today we’ll cover SLR
  – Pretty easy to learn LALR from there

• LALR(1)
  – Generally considered a good compromise between parse table size and expressiveness
  – Class for Java CUP, yacc, and bison
How does Bottom-up Parsing work?

• Already seen 1 such parser: CYK
  – Simultaneously tracked every possible parse tree
    • LR parsers work in a similar same way

• Contrast this to top-down parser
  – We know exactly where we are in the parse
  – Make predictions about what’s next
Parser State

- **Top-down parser state**
  - Current token
  - Stack of symbols
    - Represented what we expect in the rest of our descent to the leaves
  - Worked down and to the left through tree

- **Bottom-up state**
  - Also maintains a stack and token
    - Represents summary of input we’ve seen
  - Works upward and to the right through the tree
  - Also have an auxiliary state machine to help disambiguate rules

Grammar:

\[
S ::= \epsilon \\
\mid ( S ) \\
\mid [ S ]
\]
LR Derivation Order

• Let’s remember derivation orders again

Reverse  Rightmost derivation

8   1  \[ E \Rightarrow E + T \]
7   2  \[ \Rightarrow E + T * F \]
6   3  \[ \Rightarrow E + T * \text{id} \]
5   4  \[ \Rightarrow E + F * \text{id} \]
4   5  \[ \Rightarrow E + \text{id} * \text{id} \]
3   6  \[ \Rightarrow T + \text{id} * \text{id} \]
2   7  \[ \Rightarrow F + \text{id} * \text{id} \]
1   8  \[ \Rightarrow \text{id} + \text{id} * \text{id} \]
Parser Operations

• Top-down parser
  – *Scan* the next input token
  – *Push* a bunch of RHS symbols
  – *Pop* a single symbol

• Bottom-up parser
  – *Shift* an input token into a stack item
  – *Reduce* a bunch of stack items into a new parent item (on the stack)
Parser Actions: Simplified view

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id + id * id EOF</td>
<td>shift(id)</td>
</tr>
<tr>
<td>+ id * id EOF</td>
<td>reduce by F → id</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>+ id * id EOF</td>
<td>reduce by T → F</td>
</tr>
<tr>
<td>T</td>
<td>+ id * id EOF</td>
<td>reduce by E → T</td>
</tr>
<tr>
<td>E</td>
<td>+ id * id EOF</td>
<td>shift +</td>
</tr>
<tr>
<td>E + id</td>
<td>id * id EOF</td>
<td>shift id</td>
</tr>
<tr>
<td>E + F</td>
<td>* id EOF</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>E + T</td>
<td>* id EOF</td>
<td>reduce by T → F</td>
</tr>
<tr>
<td>E + T *</td>
<td>id EOF</td>
<td>shift *</td>
</tr>
<tr>
<td>E + T * id</td>
<td>EOF</td>
<td>shift id</td>
</tr>
<tr>
<td>E + T * F</td>
<td>EOF</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>E + T</td>
<td>EOF</td>
<td>reduce by T → T * F</td>
</tr>
<tr>
<td>E</td>
<td>EOF</td>
<td>reduce by E → E + T</td>
</tr>
<tr>
<td>E</td>
<td>EOF</td>
<td>accept</td>
</tr>
</tbody>
</table>
Stack Items

• Note that the previous slide was called “simplified”
• Stack elements are representative of symbols
  – Actually known as items
    • Indicate a production and a position within the production
      \[ X \rightarrow \alpha \cdot B \beta \]
    • Means
      – we are in a production of \( X \)
      – We believe we’ve parsed (arbitrary) symbol string \( \alpha \)
      – We could handle a production of \( B \)
      – After that we’ll have \( \beta \)
Stack Item Examples

• Example 1

\[ PList \rightarrow ( . \ IDList ) \]

• Example 2

\[ PList \rightarrow ( \ IDList . ) \]

• Example 3

\[ PList \rightarrow ( IDList ) . \]

• Example 4

\[ PList \rightarrow . ( IDList ) \]
• You may not know exactly which item you are parsing
• LR Parsers actually track the set of states that you could have been in

Grammar snippet

\[
\begin{align*}
  S & \rightarrow A \\
  A & \rightarrow B \\
  & \mid C \\
  B & \rightarrow D \text{ id} \\
  C & \rightarrow \text{id} E \\
  D & \rightarrow \text{id} E \\
\end{align*}
\]

\{S \rightarrow .A, A \rightarrow .B, A \rightarrow .C, \ldots\}
LR Parser FSM

Grammar G
S' → PList
PList → ( IDList )
IDList → id
IDList → IDList id

S' → . PList
PList → . ( IDList )
PList → ( IDList ) .
IDList → ( id )
IDList → IDList . id
IDList → IDList id .

PList → ( . IDList )
IDList → . id
IDList → IDList id
IDList → IDList id .

PList → IDList .
IDList → IDList . id
IDList → IDList .
IDList → IDList . id
Automaton as a table

- *Shift* corresponds to taking a terminal edge
- *Reduce* corresponds to taking a nonterminal edge

### Action table

<table>
<thead>
<tr>
<th></th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>S4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>S5</td>
<td>S6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### GoTo table

<table>
<thead>
<tr>
<th></th>
<th>PList</th>
<th>IDList</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How do we know to reduce?

- Only see terminals in the input
- Actually do reduce steps in 2 phases
  - Action table will tell us when to reduce (and how much)
  - GoTo will tell us where to... go to

<table>
<thead>
<tr>
<th>Action table</th>
<th>GoTo table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>0</td>
<td>S 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S 5</td>
</tr>
<tr>
<td>4</td>
<td>R ❸</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R ④</td>
</tr>
</tbody>
</table>

Grammar G

1. $S' \rightarrow PList$
2. $PList \rightarrow ( IDList )$
3. $IDList \rightarrow id$
4. $IDList \rightarrow IDList \ id$
## How do we know we’re done?

- Add an accept token
- Any other cell is an error

### Action table

<table>
<thead>
<tr>
<th></th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>eof</th>
<th>PList</th>
<th>IDList</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>😊</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>S</td>
<td>4</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>S</td>
<td>5</td>
<td>S 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>3</td>
<td>R</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>R</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>4</td>
<td>R</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### GoTo table

<table>
<thead>
<tr>
<th>PList</th>
<th>IDList</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

### Grammar G

1. $S' \rightarrow PList$
2. $PList \rightarrow ( IDList )$
3. $IDList \rightarrow id$
4. $IDList \rightarrow IDList \ id$
Full Parse Table Operation

Initialize stack
a = scan()
do forever
  t = top-of-stack (state) symbol
  switch action[t, a] {
  case shift s:
    push(s)
    a = scan()
  case reduce by A → alpha:
    for i = 1 to length(alpha) do pop() end
    t = top-of-stack symbol
    push(goto[t, A])
  case accept:
    return( SUCCESS )
  case error:
    call the error handler
    return( FAILURE )
  }
end do
Example Time
**Grammar G**

1. $S' \rightarrow PList$
2. $PList \rightarrow ( IDList )$
3. $IDList \rightarrow id$
4. $IDList \rightarrow IDList \cdot id$

**Current Content**

`(( id id id ) eof)`

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S 2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S 5</td>
<td>S 6</td>
</tr>
<tr>
<td>4</td>
<td>R 3</td>
<td>R 3</td>
</tr>
<tr>
<td>5</td>
<td>R 2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R 4</td>
<td>R 4</td>
</tr>
</tbody>
</table>
Seems that LR Parser works pretty great. What could possible go wrong?
LR Parser State Explosion

- Tracking sets of states can cause the size of the FSM to blow up
- The SLR and LALR variants exist to combat this explosion
- Slight modification to item and table form
Building the SLR Automaton

• Uses 2 sets
  – Closure(I)
    • What is the set of items we could be in?
    • Given I: what is the set of items that could be mistaken for I (reflexive)
  – Goto(I,X)
    • If we are in state I, where might we be after parsing X?

• Vaguely reminiscent of FIRST and FOLLOW
Closure Sets

Put I itself into Closure(I)

While there exists an item in Closure(I) of the form

\[ X \rightarrow \alpha . B \beta \]

such that there is a production \( B \rightarrow \gamma \),

and \( B \rightarrow . \gamma \) is not in Closure(I)

add \( B \rightarrow . \gamma \) to Closure(I)
GoTo Sets

Goto(I, X) = Closure({ A → α X . B | A → α . X β is in I })
**Grammar G**

\[
S' \rightarrow PList \\
PList \rightarrow (\text{IDList}) \\
IDList \rightarrow \text{id} \\
IDList \rightarrow \text{id} .\text{IDList} \text{id}
\]

**Parse Table Construction**

1. Add new start \( S' \) and \( S' \rightarrow S \)
2. Build State \( I_0 \) for Closure( \{ \( S' \rightarrow .\ S \) \})
3. Saturate FSM:
   - for each symbol \( X \) s.t. there is an item in state \( j \) containing \( . \ X \)
     - add transition from state \( j \) to state for \( \text{GoTo}(j, X) \)

**GoTo(\text{Initial})**

- Put \( \text{Closure}(I) \) in the Parse Table.
- A for \( X, \beta \) s.t. \( A \rightarrow \alpha. X \beta \in I \)
  - \( X \rightarrow \alpha. \beta \in \text{Closure}(I) \)
    - add \( B \rightarrow \gamma \) to \( \text{Closure}(I) \)
- GoTo(\( I_0 \), \( PList \)) for all items \( \alpha \rightarrow PList. \beta \)
- GoTo(\( I_0 \), \( PList \)) for all items \( A \rightarrow q \text{IDList}. \beta \)
- **Rule 1** (\( PList \rightarrow . \text{IDList} \))
  - those where \( A \rightarrow \alpha \text{IDList}. \beta \in I_0 \)
  - set to closure of the following:
    - \{ \( PList \rightarrow . \text{IDList} \) \}
    - \{ \( \text{IDList} \rightarrow . \text{id} \) \}
    - \{ \( \text{IDList} \rightarrow . \text{id} \) \}
    - \{ \( PList \rightarrow (\text{IDList}) \) \}

**GoTo(\text{Initial})**

- Only terminals after . so closure done
From FSM to parse table(s)

Need to connect the FSM back to the grammar

Grammar G:
1. $S' \rightarrow PList$
2. $PList \rightarrow ( IDList )$
3. $IDList \rightarrow id$
4. $IDList \rightarrow IDList \ id$

$S' \rightarrow . PList$
$PList \rightarrow . ( IDList )$

$PList \rightarrow ( . IDList )$
$IDList \rightarrow . id$
$IDList \rightarrow . IDList \ id$

$I_0$
$I_1$
$I_2$
$I_3$
$I_4$
$I_5$
$I_6$

$I_0$
$I_1$
$I_2$
$I_3$
$I_4$
$I_5$
$I_6$
Can Now Build Action and GoTo Tables

\[
\begin{align*}
S' & \rightarrow . PList \\
PList & \rightarrow . ( \ IDList ) \\
( & \rightarrow . PList \\
\) & \rightarrow ( . IDList ) \\
IDList & \rightarrow . \ id \\
IDList & \rightarrow . IDList \ id \\
id & \rightarrow id \\
IDList & \rightarrow id \\
IDList & \rightarrow . IDList id \\
PList' & \rightarrow ( IDList . ) \\
\) & \rightarrow \) \\
IDList & \rightarrow IDList . id \\
IDList & \rightarrow IDList id \\
\]
Building the GoTo Table

For every nonterminal $X$ if there is an $(i,j)$ edge on $X$ set $\text{GoTo}[i,X] = j$.

<table>
<thead>
<tr>
<th></th>
<th>PList</th>
<th>IDList</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Building the Action Table

• if state i includes item $A \rightarrow \alpha . \ t \beta$
  where $t$ is a terminal
  and there is an (i,j) transition on $t$
  set $\text{Action}[i,t] = \text{shift} \ j$
• If state i includes item $A \rightarrow \alpha .$
  where $A$ is not $S'$
    for each $t$ in FOLLOW($A$):
      set $\text{Action}[i,t] = \text{reduce by} \ A \rightarrow \alpha$
• if state i includes item $S \rightarrow S .$
  set $\text{Action}[i, \text{eof}] = \text{accept}$
• All other entries are error actions
### Action Table: Shift

if state $i$ includes item $A \rightarrow \alpha . t \beta$

where $t$ is a terminal

and there is an $(i,j)$ transition on $t$

set $\text{Action}[i,t] = \text{shift} \ j$

<table>
<thead>
<tr>
<th></th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>S 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S 5</td>
<td>S 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Action Table: Reduce

if state i includes item $A \rightarrow \alpha$.
where $A$ is not $S'$
for each $t$ in FOLLOW($A$):
set $\text{Action}[i,t] = \text{reduce by } A \rightarrow \alpha$

\[
\text{FOLLOW}(\text{IDList}) = \{ ), \text{id} \}
\]
\[
\text{FOLLOW}(\text{PList}) = \{ \text{eof} \}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>id</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>S 4</td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>5</td>
<td>S 6</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>3</td>
<td>R 3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>R 2</td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>4</td>
<td>R 4</td>
</tr>
</tbody>
</table>

Grammar $G$

1. $S' \rightarrow \text{PList}$
2. $\text{PList} \rightarrow ( \text{IDList} )$
3. $\text{IDList} \rightarrow \text{id}$
4. $\text{IDList} \rightarrow \text{IDList id}$

Diagram:

- $S' \rightarrow \text{PList}$
- $\text{PList} \rightarrow . ( \text{IDList} )$
- $\text{IDList} \rightarrow . \text{id}$
- $\text{IDList} \rightarrow . \text{IDList id}$
- $\text{S'} \rightarrow \text{PList}$
- $\text{S'} \rightarrow . \text{PList}$
- $\text{PList} \rightarrow . ( \text{IDList} )$
- $\text{IDList} \rightarrow . \text{id}$
- $\text{IDList} \rightarrow . \text{IDList id}$
- $\text{PList} \rightarrow ( \text{IDList} ) .$
- $\text{IDList} \rightarrow \text{id}$
- $\text{IDList} \rightarrow \text{IDList id}$
- $\text{PList} \rightarrow ( \text{IDList} . )$
- $\text{IDList} \rightarrow \text{id}$
- $\text{IDList} \rightarrow \text{IDList id}$
if state $i$ includes item $S' \rightarrow S \cdot$
set Action[$i$, $\text{eof}$] = accept
Some Final Thoughts on LR Parsing

• A bit complicated to build the parse table
  – Fortunately, algorithms exist

• Still not all powerful
  – Shift/reduce: action table cell includes S and R
  – Reduce/reduce: cell include > 1 R rule

• SDT similar to LL(1)
  – Embed SDT action numbers in action table
  – Fire off on reduce rules