LR Bottom-up Parsing
Roadmap

Last class
  – Name analysis

Previous-ish last class
  – LL(1)

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

Bottom-Up parsing

- 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 \( S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow w \) 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 \( w \).

- A grammar is LR(k) if for every derivation step, \( A \rightarrow \beta \) 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 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 parser state
- Also maintains a stack and token
  • Represents summary of input we’ve seen
- Works upward and to the right through the tree
- Also has an auxiliary state machine to help disambiguate rules

Grammar

\[
S ::= \varepsilon \\
\mid ( \ S \ ) \\
\mid [ \ S ]
\]

Stack

[ \\
S \\
] \\
) \\
EOF
LR Derivation Order

Let’s remember derivation orders again

<table>
<thead>
<tr>
<th>Reverse</th>
<th>Rightmost derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1. $E \Rightarrow E + T$</td>
</tr>
<tr>
<td>7</td>
<td>2. $\Rightarrow E + T \ast F$</td>
</tr>
<tr>
<td>6</td>
<td>3. $\Rightarrow E + T \ast \text{id}$</td>
</tr>
<tr>
<td>5</td>
<td>4. $\Rightarrow E + F \ast \text{id}$</td>
</tr>
<tr>
<td>4</td>
<td>5. $\Rightarrow E + \text{id} \ast \text{id}$</td>
</tr>
<tr>
<td>3</td>
<td>6. $\Rightarrow T + \text{id} \ast \text{id}$</td>
</tr>
<tr>
<td>2</td>
<td>7. $\Rightarrow F + \text{id} \ast \text{id}$</td>
</tr>
<tr>
<td>1</td>
<td>8. $\Rightarrow \text{id} + \text{id} \ast \text{id}$</td>
</tr>
</tbody>
</table>
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 + id * id EOF</td>
<td>shift(id)</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>+ id * id EOF</td>
<td>reduce by F → id</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 +</td>
<td>id * id EOF</td>
<td>shift id</td>
</tr>
<tr>
<td>E + id</td>
<td>* id EOF</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>E + F</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 *</td>
<td>id EOF</td>
<td>shift id</td>
</tr>
<tr>
<td>E + T * id</td>
<td>reduce by F → id</td>
<td></td>
</tr>
<tr>
<td>E + T * F</td>
<td>reduce by T → T * F</td>
<td></td>
</tr>
<tr>
<td>E + T</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>

```
Input: id + id * id EOF
Stack: id + id * id EOF
Action: shift(id)
```
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
    \[ X \rightarrow \alpha . B \beta \]
  • 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

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

Example 2

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

Example 3

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

Example 4

\[ PList \rightarrow . ( \text{IDList} ) \]
Stack Item State

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**

\[ S' \rightarrow PList \]

\[ PList \rightarrow ( \text{IDList} ) \]

\[ IDList \rightarrow \text{id} \]

\[ IDList \rightarrow IDList \text{id} \]

\[ PList \rightarrow \text{id} . \]

\[ IDList \rightarrow \text{id} . \]

\[ IDList \rightarrow IDList \text{id} . \]
Automaton as a table

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

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

Shift and go to state 6
How do we know when to reduce?

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

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

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
How do we know we’re done?

Add an accept token
Any other cell is an error

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. \( IDList \rightarrow IDList \ id \)

<table>
<thead>
<tr>
<th>Input</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>1</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></td>
<td>S 4</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>S 5</td>
<td>S 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>3</td>
<td>R 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>R 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>4</td>
<td>R 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Seems that LR Parser works 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(s,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 form

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

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 $G$

- $S' \rightarrow PList$
- $PList \rightarrow ( \text{IDList} )$
- $IDList \rightarrow \text{id}$
- $IDList \rightarrow IDList \text{id}$

**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 a item in state $j$ containing $.$ $X$
     add transition from state $j$ to state for $\text{GoTo}(j, X)$

**Closure**

- $\text{Closure}(I)$
  - Put $I$ in $\text{Closure}(I)$
  - Repeat for $X \rightarrow \alpha \cdot B \cdot \beta \epsilon I$
    - $B \rightarrow \gamma$ s.t. $\exists B \rightarrow \gamma$, add $B \rightarrow . \gamma$ to $\text{Closure}(I)$
  - until saturation

**GoTo**

- $\text{GoTo}(I_0, PList)$
  - all items $A \rightarrow \alpha \cdot PList \cdot \beta$
    - all items $A \rightarrow \alpha \cdot \text{idList} \cdot \beta$
    - those where $\text{IDList} \rightarrow . \beta \epsilon I_0$
      for $[1] PList \rightarrow ( \text{IDList} )$
      set to closure of the following:
      - for $[1] PList \rightarrow ( \text{IDList} )$ is in $I_2$
        - $\{ PList \rightarrow ( \text{IDList} ) \}$ in $I_2$
      - for $[2] PList \rightarrow \text{IDList} \cdot \text{IDList}$ is in $I_2$
        - $\{ \text{IDList} \rightarrow \text{id} \}$ in $I_2$
        - $\{ \text{IDList} \rightarrow . \}$ in $I_2$
  - those where $A \rightarrow \alpha \cdot \text{IDList} \beta \epsilon I_0$
    - set to closure of the following:
      - $\{ \text{IDList} \rightarrow \text{id} \}$ in $I_0$

Only terminals after $.$ so closure done

Done with closure, and $\text{GoTo}$
From FSM to parse table(s)

Grammar G

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

Need to connect the FSM back to the grammar
Can Now Build Action and GoTo Tables

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

\[ ( \) \]

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

\[ \text{id} \]

\[ \text{PList} \rightarrow ( ) \]

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

\[ \text{IDList} \rightarrow \text{id} \]

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

\[ \text{id} \]

\[ \text{IDList} \rightarrow \text{IDList id} . \]
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></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></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 \( \text{FOLLOW}(A) \):
- 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 $A \rightarrow \alpha \cdot t \beta$
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></td>
<td>S 4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>S 5</td>
<td></td>
<td>S 6</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>
Grammar G

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

**Action Table: Reduce**

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

$\text{FOLLOW}(IDList) = \{ , \text{id} \}$

$\text{FOLLOW}(PList) = \{ \text{eof} \}$

<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></td>
<td>S 5</td>
<td>S 6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>R ③</td>
<td>R ③</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>R ②</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>R ④</td>
<td>R ④</td>
<td></td>
</tr>
</tbody>
</table>
if state $i$ includes item $S' \rightarrow S$. set $\text{Action}[i,\text{eof}] = \text{accept}$

<table>
<thead>
<tr>
<th>0</th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>😊</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>S 5</td>
<td>S 6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>R 3</td>
<td>R 3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>R 2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>R 4</td>
<td>R 4</td>
</tr>
</tbody>
</table>
Some Final Thoughts on LR Parsing

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

Still not as powerful as CYK
  – 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