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

– Describe 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., create a 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
– LR parsers more powerful than LL parsers: LL(1) ⊆ LR(1)

Disadvantages
– More complex parser generation
– Larger parse tables

Warning! Subtle point ahead!
When talking about families of languages:
LL(1)_{\text{LANG}} \subseteq LR(1)_{\text{LANG}}

However, that is not true of grammars: You can have an LL(1) grammar for language \mathcal{L} that is not an LR(1) grammar; however, there is always another grammar for \mathcal{L} that is LR(1)

I.e., rightmost derivation in reverse
Rightmost derivation generates A last
Bottom-up parser identifies A first
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

Recognizable by a deterministic PDA
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 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
  - Represents what is expected in the rest of the input
- Parser works down and to the right through the tree

**Bottom-up parser state**
- Also maintains a stack and current token
  - Represents summary of input seen so far
- Works upward and to the right through the tree
- Also has an auxiliary state machine to help disambiguate rules

**Grammar**

\[
S ::= \epsilon \\
| ( \ S \ ) \\
| [ \ S \ ]
\]

**Stack**

- EOF
- )
- ]
- S
- [

**Current**

[ ]
A Snapshot of a Predictive Parser

The structure already seen

The structure that the parser expects to build

"Work to do" Stack

The structure already seen

Input: Already processed current Not yet seen

S

B

A

A

D

C

t

eof

t

u

eof

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  $\Rightarrow E + T * F$</td>
</tr>
<tr>
<td>6</td>
<td>3  $\Rightarrow E + T * id$</td>
</tr>
<tr>
<td>5</td>
<td>4  $\Rightarrow E + F * id$</td>
</tr>
<tr>
<td>4</td>
<td>5  $\Rightarrow E + id * id$</td>
</tr>
<tr>
<td>3</td>
<td>6  $\Rightarrow T + id * id$</td>
</tr>
<tr>
<td>2</td>
<td>7  $\Rightarrow F + id * id$</td>
</tr>
<tr>
<td>1</td>
<td>8  $\Rightarrow id + id * id$</td>
</tr>
</tbody>
</table>
Parser Operations

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

Bottom-up parser
- *Shift* an input token into a stack item
- *Reduce* a bunch of stack items into a new parent item (and push the parent 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>reduce by F → id</td>
</tr>
<tr>
<td>id</td>
<td>+ id * id EOF</td>
<td>reduce by T → F</td>
</tr>
<tr>
<td>F</td>
<td>+ id * id EOF</td>
<td>reduce by E → T</td>
</tr>
<tr>
<td>T</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>EOF</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>E + T * F</td>
<td>EOF</td>
<td>reduce by T → T * F</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>
Stack Items

The previous slide was simplified

Stack elements represent more than just symbols

– Stack elements known as “items”
  • Indicates a production and a position within the production

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

• This item indicates
  – We are in a production of X
  – We (might) have parsed a string in L(\(\alpha\))
  – We could now handle a production of B
  – After that we’ll have a string in L(\(\beta\))
Stack-Item Examples

Example 1

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

Example 2

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

Example 3

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

Example 4

\[ PList \rightarrow . ( IDList ) \]
Stack-Item States

The parser may not know exactly which item it is parsing.

LR parsers actually track the set of items that it could be in.

Grammar snippet:

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

\{S\rightarrow . A, A\rightarrow . B, A\rightarrow . C, ...\}
Grammar G

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

LR Parser FSM
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>
<th>PList</th>
<th>IDList</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S2</td>
<td></td>
<td>id</td>
<td>eof</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></td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>S5</td>
<td>S6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Shift and go to state 6**
How do we know when 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

**Grammar G**

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

<table>
<thead>
<tr>
<th>Action table</th>
<th>GoTo table</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) id eof</td>
<td></td>
</tr>
<tr>
<td>0  S 2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4 R 3 R 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6 R 4 R 4</td>
<td></td>
</tr>
</tbody>
</table>
How do we know we’re done?

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

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

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 $A \rightarrow \alpha$”:
      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

Whenever “reduce by $A \rightarrow \alpha$” happens, the parser’s state corresponds to an itemset that contains the item “$A \rightarrow \alpha$.” Each symbol of $\alpha$ has been shifted onto the stack; therefore, each symbol in $\alpha$ needs to be popped off the stack.
Example Time
Grammar G
1. $S' \rightarrow PList$
2. $PList \rightarrow ( IDList )$
3. $IDList \rightarrow id$
4. $IDList \rightarrow IDList id$

Current Element
(id id id) eof

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Action</th>
<th>State</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>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></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></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R 2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R 4</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(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 $X \rightarrow \alpha \cdot B \beta$

such that there is a production $B \rightarrow \gamma$

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

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

Goto(I, X) =

Closure(\{ A \rightarrow \alpha X . B \mid A \rightarrow \alpha . X \beta \text{ is in I} \})
Parse Table Construction

1: Add new start $S'$ and $S' \rightarrow S$

2: Build State $I_0$ for Closure($\{S' \rightarrow . \} \}$

3: Saturate FSM: for each symbol $X$ s.t. there is an item in state $j$ containing $X$ add transition from state $j$ to the state whose items are GoTo($j$, $X$)

---

**Grammar $G$**

$S' \rightarrow \text{PList}$

$\text{PList} \rightarrow (\text{IDList})$

$\text{IDList} \rightarrow \text{id}$

$\text{IDList} \rightarrow \text{id} \cdot \text{IDList}$

---

**Parse Table Construction**

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

---

**Closure**

Closure{ $S' \rightarrow . \}$ =

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

---

**GoTo Table Construction**

- GoTo($I_0$, $\text{id}$)
- GoTo($I_0$, $\text{IDList}$)
- GoTo($I_0$, $\text{PList}$)
- GoTo($I_0$, $\text{S'}$)

---

**Items**

- $\text{IDList} \rightarrow . \text{id}$
- $\text{IDList} \rightarrow . \text{IDList} \text{id}$

---

**Done with closure, and GoTo done**
From FSM to parse table(s)

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

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

$I_0$

$I_2$

$I_4$

$I_1$

$I_3$

$I_5$

$I_6$

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

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

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

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

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

\[ \text{IDList} \rightarrow \text{IDList id} . \]
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 $A \rightarrow \alpha \cdot t \beta$
- where $t$ is a terminal
- and there is an $(i,j)$ transition on $t$
- set $\text{Action}[i, t] = \text{shift} \ j$

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
if state i includes item $A \rightarrow \alpha . \mathbf{t} \mathbf{\beta}$
where $\mathbf{t}$ is a terminal
and there is an (i,j) transition on $\mathbf{t}$
set $\text{Action}[i,\mathbf{t}] = \text{shift j}$

<table>
<thead>
<tr>
<th>$S'$</th>
<th>$\mathbf{PList}$</th>
<th>$\mathbf{id}$</th>
<th>$\text{eof}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>S 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S 5</td>
<td>S 6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</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 $\text{FOLLOW}(A)$:
set $\text{Action}[i,t] = \text{reduce by } A \rightarrow \alpha$

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

<table>
<thead>
<tr>
<th></th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
<td>2</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</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>S</td>
<td>5</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>3</td>
<td>R</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>4</td>
<td>R</td>
<td>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 $\text{IDList} \rightarrow \text{IDList id}$
if state i includes item $S' \rightarrow S$.
set Action[i, eof] = accept

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

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>😊</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>S 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S 5</td>
<td>S 6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R 3</td>
<td>R 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>R 2</td>
</tr>
<tr>
<td>6</td>
<td>R 4</td>
<td>R 4</td>
<td></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