Bottom-Up Parsing

Lecture 11-12 (From slides by G. Necula & R. Bodik)

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Most common form is LR parsing
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation

An Introductory Example

- LR parsers don’t need left-factored grammars and can also handle left-recursive grammars
- Consider the following grammar:
  \[ E \rightarrow E + (E) | \text{int} \]
  - Why is this not LL(1)?
- Consider the string: \[ \text{int} + (\text{int}) + (\text{int}) \]

The Idea

- LR parsing reduces a string to the start symbol by inverting productions:
  \[ \text{str} \leftarrow \text{input string of terminals} \]
  while \( \text{str} \neq S \):
    - Identify first \( \beta \) in \( \text{str} \) such that \( A \rightarrow \beta \) is a production and \( S \rightarrow^* \alpha A \gamma \rightarrow \alpha \beta \gamma = \text{str} \)
    - Replace \( \beta \) by \( A \) in \( \text{str} \) (so \( \alpha A \gamma \) becomes new \( \text{str} \))
    - Such \( \alpha \beta \)'s are called handles

A Bottom-up Parse in Detail (1)

\[ \text{int} \ast (\text{int}) \ast (\text{int}) \]

A Bottom-up Parse in Detail (2)

\[ \begin{align*}
E & \rightarrow \text{int} \ast (\text{int}) \\
E & \rightarrow (\text{int}) \ast (\text{int}) \\
E & \rightarrow (\text{int}) \ast (\text{int}) \\
E & \leftarrow \text{int} \ast (\text{int}) \ast (\text{int})
\end{align*} \]
A Bottom-up Parse in Detail (3)

\[
\begin{align*}
\text{int} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{E}) &+ (\text{int})
\end{align*}
\]

\[
\text{E} \quad \text{E} \\
| \quad | \\
\text{int} + \{ \text{int} \} + \{ \text{int} \}
\]

A Bottom-up Parse in Detail (4)

\[
\begin{align*}
\text{int} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{E}) &+ (\text{int}) \\
\text{E} &+ (\text{int})
\end{align*}
\]

\[
\text{E} \quad \text{E} \\
| \quad | \\
\text{int} + \{ \text{int} \} + \{ \text{int} \}
\]

A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{E}) &+ (\text{int}) \\
\text{E} &+ (\text{int})
\end{align*}
\]

\[
\text{E} \quad \text{E} \\
| \quad | \\
\text{int} + \{ \text{int} \} + \{ \text{int} \}
\]

A Bottom-up Parse in Detail (6)

\[
\begin{align*}
\text{int} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{int}) &+ (\text{int}) \\
\text{E} + (\text{E}) &+ (\text{int}) \\
\text{E} &+ (\text{int})
\end{align*}
\]

\[
\text{E} \quad \text{E} \\
| \quad | \\
\text{int} + \{ \text{int} \} + \{ \text{int} \}
\]

Where Do Reductions Happen

Because an LR parser produces a reverse rightmost derivation:

- If \( \alpha \beta \gamma \) is step of a bottom-up parse with handle \( \alpha \beta \)
- And the next reduction is by \( A \rightarrow \beta \)
- Then \( \gamma \) is a string of terminals!

... Because \( \alpha A \gamma \rightarrow \alpha \beta \gamma \) is a step in a right-most derivation

Intuition: We make decisions about what reduction to use after seeing all symbols in handle, rather than before (as for LL(1))

Notation

- Idea: Split the string into two substrings
  - Right substring (a string of terminals) is as yet unexamined by parser
  - Left substring has terminals and non-terminals

- The dividing point is marked by a \( I \)
  - The \( I \) is not part of the string
  - Marks end of next potential handle

- Initially, all input is unexamined: \( x_1 x_2 \ldots x_n \)
Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:
  
  Shift: Move one place to the right, shifting a terminal to the left string
  
  \[ E \rightarrow (E) \Rightarrow E \rightarrow (E') \]
  
  Reduce: Apply an inverse production at the handle.
  
  If \[ E \rightarrow E + (E) \] is a production, then
  
  \[ E + (E + (E)) \Rightarrow E + (E') \]

Shift-Reduce Example

1. \[ \text{int} \rightarrow (\text{int}) \Rightarrow \text{shift} \]
2. \[ \text{int} \rightarrow (\text{int}) \Rightarrow \text{red. } E \rightarrow \text{int} \]

\[ \text{int} + (\text{int}) + (\text{int}) \]

\[ \uparrow \]

Shift-Reduce Example

1. \[ \text{int} \rightarrow (\text{int}) \Rightarrow \text{shift} \]
2. \[ \text{int} \rightarrow (\text{int}) \Rightarrow \text{red. } E \rightarrow \text{int} \]
3. \[ E \rightarrow \text{int} \Rightarrow \text{shift 3 times} \]

\[ E \]

\[ \text{int} + (\text{int}) + (\text{int}) \]

\[ \uparrow \]

Shift-Reduce Example

1. \[ \text{int} \rightarrow (\text{int}) \Rightarrow \text{shift} \]
2. \[ \text{int} \rightarrow (\text{int}) \Rightarrow \text{red. } E \rightarrow \text{int} \]
3. \[ E \rightarrow \text{int} \Rightarrow \text{shift 3 times} \]
4. \[ E \rightarrow \text{int} \Rightarrow \text{red. } E \rightarrow \text{int} \]

\[ E \]

\[ \text{int} + (\text{int}) + (\text{int}) \]

\[ \uparrow \]
Shift-Reduce Example

\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ int \]
\[ int \]
\[ int + (int) + (int) \]

\[ \rightarrow \]

Shift-Reduce Example

\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ int \]
\[ int \]
\[ int + (int) + (int) \]

\[ \rightarrow \]

Shift-Reduce Example

\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ int \]
\[ int + (int) + (int) \]

\[ \rightarrow \]

Shift-Reduce Example

\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ int + (int) + (int) \]

\[ \rightarrow \]

Shift-Reduce Example

\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ E + (E) \]
\[ int + (int) + (int) \]

\[ \rightarrow \]
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 from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Key Issue: When to Shift or 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 up to potential handle
  - DFA alphabet consists of terminals and nonterminals
  - DFA recognizes complete handles
- We run the DFA on the stack and we examine the resulting state \( X \) and the token \( tok \) after \( l \)
  - If \( X \) has a transition labeled \( tok \) then shift
  - If \( X \) is labeled with "A \( \rightarrow \) B on tok" then reduce

LR(1) Parsing. An Example

```
int E \rightarrow int
E \rightarrow int on $, +
E \rightarrow (E) on $, +
E \rightarrow E + (E) on $, +
E \rightarrow int
E \rightarrow (E) on $, +
```

Representing the DFA

- Parsers represent the DFA as a 2D table
  - As for table-driven lexical analysis
  - Lines correspond to DFA states
  - Columns correspond to terminals and non-terminals
  - In classical treatments, columns are split into:
    - Those for terminals: action table
    - Those for non-terminals: goto table

Representing the DFA. Example

The table for a fragment of our DFA:

<table>
<thead>
<tr>
<th></th>
<th>int + (int) + (int)$</th>
<th>shift</th>
<th>int + (int)</th>
<th>int $</th>
<th>E \rightarrow int</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>s6</td>
<td>6</td>
<td>7</td>
<td>E \rightarrow (E) on $, +</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td>8</td>
<td>s7</td>
<td>r_E \rightarrow int</td>
</tr>
<tr>
<td>5</td>
<td>s4</td>
<td>s3</td>
<td>7</td>
<td>s6</td>
<td>E \rightarrow E=E(E)</td>
</tr>
<tr>
<td>6</td>
<td>s3</td>
<td>s2</td>
<td>6</td>
<td>s8</td>
<td>E \rightarrow E=E(E)</td>
</tr>
</tbody>
</table>

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- So record, for each stack element, state of the DFA after that state
- LR parser maintains a stack \((\text{sym}_1, \text{state}_1) \ldots (\text{sym}_n, \text{state}_n)\)
  - \(\text{state}_n\) is the final state of the DFA on \(\text{sym}_1 \ldots \text{sym}_n\)
**The LR Parsing Algorithm**

Let $I = w_1w_2...w_n$ be initial input
Let $j = 1$
Let DFA state $0$ be the start state
Let stack = (dummy, 0)

repeat
  case action [top_state(stack), $I[j]$] of
    shift $k$: push ($I[j], k$), $j += 1$
    reduce $X \rightarrow \alpha$:
      pop $\alpha$ pairs,
      push ($X$, Goto [top_state(stack), $X$])
    accept: halt normally
    error: halt and report error

---

**LR Parsing Notes**

- Can be used to parse more grammars than LL
- Most programming languages grammars are LR
- Can be described as a simple table
- There are tools for building the table
- How is the table constructed?

---

**To Be Done**

- Review of bottom-up parsing
- Computing the parsing DFA
- Using parser generators

---

**Bottom-up Parsing (Review)**

- A bottom-up parser rewrites the input string to the start symbol
- The state of the parser is described as $\alpha \gamma$
  - $\alpha$ is a stack of terminals and non-terminals
  - $\gamma$ is the string of terminals not yet examined
- Initially: $I x_1 x_2 ... x_n$

---

**The Shift and Reduce Actions (Review)**

- Recall the CFG: $E \rightarrow \text{int} | E + (E)$
- A bottom-up parser uses two kinds of actions:
  - **Shift** pushes a terminal from input on the stack
    $E + (\text{int}) \Rightarrow E + (\text{int})$
  - **Reduce** pops 0 or more symbols from the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
    $E + (E + (E)) \Rightarrow E + (E + (E))$

---

**Key Issue: When to Shift or Reduce?**

- Idea: use a finite automaton (DFA) to decide when to shift or reduce
  - The 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 $X$ and the token $tok$ after $I$
  - If $X$ has a transition labeled $tok$ then shift
  - If $X$ is labeled with "$A \rightarrow \beta$ on tok" then reduce
LR(1) Parsing. An Example

Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse
  - What non-terminal we are looking for
  - What productions we are looking for
  - What we have seen so far from the rhs

Parsing Contexts

- Consider the state:
  \[ E \rightarrow \alpha \beta, a \]
    - We are trying to find an \( E \) followed by an \( a \), and
    - \( \alpha \) is already on top of the stack
    - Thus we need to see next a prefix derived from \( \beta a \)

Note

- The symbol \( I \) was used before to separate the stack from the rest of input
  - \( \alpha \mid \gamma \), where \( \alpha \) is the stack and \( \gamma \) is the remaining string of terminals
- In LR(1) items \( \cdot \) is used to mark a prefix of a production rhs:
  \[ X \rightarrow \alpha \beta, a \]
    - Here \( \beta \) might contain non-terminals as well
    - In both case the stack is on the left

LR(1) Items

- An LR(1) item is a pair:
  \[ X \rightarrow \alpha \beta, a \]
    - \( X \rightarrow \alpha \beta \) is a production
    - \( \alpha \) is a terminal (the lookahead terminal)
    - LR(1) means 1 lookahead terminal

Convention

- We add to our grammar a fresh new start symbol \( S \) and a production \( S \rightarrow E \)
  - Where \( E \) is the old start symbol
  - No need to do this if \( E \) had only one production
- The initial parsing context contains:
  \[ S \rightarrow \cdot, E, \$
    - Trying to find an \( S \) as a string derived from \( E \$
    - The stack is empty
LR(1) Items (Cont.)

- In context containing $E \rightarrow E + (E) +$
  - If $+$ follows then we can perform a shift to context containing $E \rightarrow E + (E) +$
- In context containing $E \rightarrow E + (E) +$
  - We can perform a reduction with $E \rightarrow E + (E)$
  - But only if a $+$ follows

The Closure Operation

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

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

The DFA Transitions

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

Transition(State, y)
$\text{Items} \leftarrow \emptyset$
for each $[X \rightarrow \alpha \gamma b], b] \in \text{State}$
  add $[X \rightarrow \alpha \gamma b], b]$ to $\text{Items}$
return $\text{Closure(Items)}$
Constructing the Parsing DFA. Example.

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\rightarrow E \cdot $</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\rightarrow E \cdot (E) \cdot $, $\rightarrow E \cdot $ $</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$\rightarrow E \cdot $</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>$\rightarrow E \cdot (E) \cdot $, $\rightarrow E \cdot $ $</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>$\rightarrow E \cdot (E) \cdot $, $\rightarrow E \cdot (E) \cdot $ $</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>$\rightarrow E \cdot (E) \cdot $, $\rightarrow E \cdot $ $</td>
<td>0</td>
</tr>
</tbody>
</table>

LR Parsing Tables. Notes

- Parsing tables (i.e. the DFA) can be constructed automatically for a CFG
- But we still need to understand the construction to work with parser generators
  - E.g., they report errors in terms of sets of items
- What kind of errors can we expect?

Shift/Reduce Conflicts

- If a DFA state contains both $[X \rightarrow a \cdot a \cdot b, a]$ and $[Y \rightarrow \gamma \cdot a]$
- Then on input "a" we could either
  - Shift into state $[X \rightarrow a \cdot a \cdot b, a]$ or
  - Reduce with $Y \rightarrow \gamma$
- This is called a shift-reduce conflict

More Shift/Reduce Conflicts

- Consider the ambiguous grammar $E \rightarrow E \cdot E \| E \cdot E \| \text{int}$
- We will have the states containing $[E \rightarrow E \cdot E \cdot E, \cdot]$, $[E \rightarrow E \cdot E \cdot E, \cdot]$
- Again we have a shift/reduce on input $*$
  - We need to reduce ($^*$ binds more tightly than $\cdot$)
  - Solution: declare the precedence of $^*$ and $\cdot$

More Shift/Reduce Conflicts

- In bison declare precedence and associativity of terminal symbols:
  - $\text{left} = *$
  - Precedence of a rule = that of its last terminal
  - See bison manual for ways to override this default
- Resolve shift/reduce conflict with a shift if:
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative
Using Precedence to Solve S/R Conflicts

- Back to our example:
  \[ E \rightarrow E * E, + \]
  \[ E \rightarrow E + E, + \]

- Will choose reduce because precedence of rule \( E \rightarrow E * E \) is higher than of terminal +

Using Precedence to Solve S/R Conflicts

- Same grammar as before
  \[ E \rightarrow E + E \ | \ E * E \ | \ \text{int} \]

- We will also have the states
  \[ E \rightarrow E * E, + \]
  \[ E \rightarrow E + E, + \]

- Now we also have a shift/reduce on input +
  - We choose reduce because \( E \rightarrow E + E \) and + have the same precedence and + is left-associative

Reduce/Reduce Conflicts

- If a DFA state contains both \[ X \rightarrow \alpha, A \] and \[ Y \rightarrow \beta, A \]
  - Then on input "a" we don’t know which production to reduce

- This is called a reduce/reduce conflict

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar

- Example: a sequence of identifiers
  \[ S \rightarrow \epsilon \ | \ id \ | \ id S \]

- There are two parse trees for the string id
  \[ S \rightarrow id \]
  \[ S \rightarrow id S \rightarrow id \]

- How does this confuse the parser?

More on Reduce/Reduce Conflicts

- Consider the states
  \[ S' \rightarrow id, $ \]
  \[ S \rightarrow id \ | \ S, $ \]
  \[ S \rightarrow \epsilon, \]
  \[ S \rightarrow id \ | \ S, $ \]
  \[ S \rightarrow id S, $ \]

- Reduce/reduce conflict on input $ $
  \[ S' \rightarrow S \rightarrow S \rightarrow id \]
  \[ S \rightarrow S \rightarrow id S \rightarrow id \]

- Better rewrite the grammar:
  \[ S \rightarrow \epsilon \ | \ id S \]
Using Parser Generators

- Parser generators construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars (and is provided as a library function)
- But most parser generators do not construct the DFA as described before
  - Because the LR(1) parsing DFA has 1000s of states even for a simple language

LR(1) Parsing Tables are Big

- But many states are similar, e.g.
  ![LR(1) Parsing Tables are Big](image)
- Idea: merge the DFA states whose items differ only in the lookahead terminals
  - We say that such states have the same core
  - We obtain

The Core of a Set of LR Items

- Definition: The core of a set of LR items is the set of first components without the lookahead terminals
- Example: the core of
  \[\{ [X \rightarrow \alpha\beta, b], [Y \rightarrow \gamma\delta, d]\}\]
  is
  \[\{X \rightarrow \alpha\beta, Y \rightarrow \gamma\delta\}\]

LALR States

- Consider for example the LR(1) states
  \[\{[X \rightarrow \alpha^*, a], [Y \rightarrow \beta^*, c]\}\]
  \[\{[X \rightarrow \alpha^*, b], [Y \rightarrow \beta^*, d]\}\]
- They have the same core and can be merged
- And the merged state contains:
  \[\{[X \rightarrow \alpha^*, a/b], [Y \rightarrow \beta^*, c/d]\}\]
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

A LALR(1) DFA

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors

Conversion LR(1) to LALR(1). Example.
The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states
  \(\{[X \rightarrow \alpha^*, a], [Y \rightarrow \beta^*, b]\}\)
  \(\{[X \rightarrow \alpha^*, b], [Y \rightarrow \beta^*, a]\}\)
- And the merged LALR(1) state
  \(\{[X \rightarrow \alpha^*, a/b], [Y \rightarrow \beta^*, a/b]\}\)
- Has a new reduce-reduce conflict
- In practice such cases are rare

LALR vs. LR Parsing

- LALR languages are not natural
  - They are an efficiency hack on LR languages
- But any reasonable programming language has a LALR(1) grammar
- LALR(1) has become a standard for programming languages and for parser generators

A Hierarchy of Grammar Classes

Notes on Parsing

- Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - We use LALR(1) parser generators