Bottom-up Parsing (Review)

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

The Shift and Reduce Actions (Review)

Recall the CFG: $E \rightarrow E \; + \; (E) \mid \text{int}$

A bottom-up parser uses two kinds of actions:

- **Shift** pushes a terminal from input on the stack.
  - $E \; + \; (\text{int}\; \text{int}) \; \Rightarrow \; E \; + \; (\text{int}\; \text{int})$

- **Reduce** pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS).
  - $E \; + \; (E\; + \; (E\; \text{int})) \; \Rightarrow \; E \; + \; (E\; \text{int})$

Outline

- Review of bottom-up parsing
- Computing the parsing DFA
- Using parser generators
Key Issue: When to Shift or Reduce?

- Idea: use a deterministic 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 → β on tok” then reduce

Representing the DFA

- Parsers represent the DFA as a 2D table
  (Recall table-driven lexical analysis)
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: the action table
  - Those for non-terminals: the goto table

LR(1) Parsing: An Example

The table for a fragment of our DFA:

```
<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>+</th>
<th>( )</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td>g6</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>rE→int</td>
<td>rE→int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s8</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>rE→E+(E)</td>
<td>rE→E+(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Int $E$ is shift and goto state $k$
$r_x \rightarrow_\alpha$ is reduce
$g_k$ is goto state $k$
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

• To avoid this, we remember for each stack element on which state it brings the DFA

• LR parser maintains a stack
  $$\langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle$$
  state\(_k\) is the final state of the DFA on sym\(_1 \ldots \text{sym}_k\)

Key Issue: 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(0) Items

• An LR(0) item is a production with a "I" somewhere on the RHS

• The LR(0) items for T → (E) are
  $$T \rightarrow \mathbf{I} (E)$$
  $$T \rightarrow (\mathbf{I} E)$$
  $$T \rightarrow (E \mathbf{I})$$
  $$T \rightarrow (E) \mathbf{I}$$

• The only LR(0) item for X → ε is X → I
**LR(0) Items: Intuition**

- An item \([X → α \ I β]\) says that the parser
  - is looking for an \(X\)
  - has an \(α\) on top of the stack
  - expects to find a string derived from \(β\) next in the input

- Notes:
  - \([X → α \ I αβ]\) means that \(a\) should follow
    - Then we can shift it and still have a viable prefix
  - \([X → α \ I]\) means that we could reduce \(X\)
    - But this is not always a good idea!

**LR(1) Items**

- An LR(1) item is a pair:
  \(X → α \ I β, a\)
  - \(X → αβ\) is a production
  - \(a\) is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- \([X → α \ I β, a]\) describes a context of the parser
  - We are trying to find an \(X\) followed by an \(a\), and
  - We have (at least) \(α\) already on top of the stack
  - Thus we need to see next a prefix derived from \(βa\)

**Note**

- The symbol \(I\) was used before to separate the stack from the rest of input
  - \(α \ I γ\), where \(α\) is the stack and \(γ\) is the remaining string of terminals
- In items, \(I\) is used to mark a prefix of a production RHS:
  \(X → α \ I β, a\)
  - Here \(β\) might contain non-terminals as well
- In either case the stack is on the left of \(I\)

**Convention**

- We add to our grammar a fresh new start symbol \(S\) and a production \(S → E\)
  - Where \(E\) is the old start symbol
- The initial parsing context contains:
  \(S → I E, $\)
  - Trying to find an \(S\) as a string derived from \(E$\)
  - The stack is empty
LR(1) Items (Cont.)

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

The Closure Operation

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

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

LR(1) Items (Cont.)

• Consider the item
  \( E \to E + \{ E \} , + \)
• We expect a string derived from \( E \) +
• Our example has two productions for \( E \)
  \( E \to \text{int} \) and \( E \to E + \{ E \} \)
• We describe this by extending the context with two more items:
  \( E \to \text{int} , ) \)
  \( E \to \{ E \} + ( E ) , ) \)

Constructing the Parsing DFA (1)

• Construct the start context:
  \( E \to E + \{ E \} \{ E \} , + \)

• We abbreviate as:
  \( E \to \{ E \} + ( E ) \{ E \} , + \)

• Construct the start context:
  \( E \to E + \{ E \} , + \)

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

  • We abbreviate as:
    \[
    \begin{align*}
    S & \to \text{int} , $
    E & \to \text{int} + ( E ) , + \\
    E & \to \text{int} , + \\
    \end{align*}
    \]
Constructing the Parsing DFA (2)

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

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

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

• And now the transitions …

The DFA Transitions

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

\[
\text{Transition(State, y)} \\
\text{Items} = \emptyset \\
\text{for each } [X \rightarrow \alpha \ I \ y \beta, b] \text{ in State} \\
\text{add } [X \rightarrow \alpha y \ I \ \beta, b] \text{ to Items} \\
\text{return Closure(Items)}
\]

Constructing the Parsing DFA: Example

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
  \([X \rightarrow \alpha I a \beta, b]\) and \([Y \rightarrow \gamma I, a]\)

• Then on input “a” we could either
  - Shift into state \([X \rightarrow \alpha a \beta, b]\), or
  - Reduce with \(Y \rightarrow \gamma\)

• This is called a \textit{shift-reduce conflict}

More Shift/Reduce Conflicts

• Consider the ambiguous grammar
  \(E \rightarrow E + E | E * E | \text{int}\)

• We will have the states containing
  \([E \rightarrow E * I E, +]\) \quad \([E \rightarrow E * E I, +]\)
  \([E \rightarrow I E + E, +] \Rightarrow E\) \quad \([E \rightarrow E I + E, +]\)

  \[\ldots\]

• Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Recall solution: declare the precedence of * and +

Shift/Reduce Conflicts

• Typically due to ambiguities in the grammar
• Classic example: the dangling else
  \(S \rightarrow \text{if E then } S \mid \text{if E then } S \text{ else } S \mid \text{OTHER}\)

• Will have a DFA state containing
  \([S \rightarrow \text{if E then } S I, \text{ else}]\)
  \([S \rightarrow \text{if E then } S I \text{ else } S, \text{ x}]\)

• If else follows then we can shift or reduce
• Default (yacc, ML-yacc, bison, etc.) is to shift
  - Default behavior is as needed in this case

More Shift/Reduce Conflicts

• In yacc declare precedence and associativity:
  \%
left +
%
left *

• Precedence of a rule = that of its last terminal
  See yacc manual for ways to override this default

• Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - 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:
  
  \[ E \rightarrow E \times I E, + \]  
  \[ E \rightarrow E \times E I, + \]  
  \[ E \rightarrow I E + E, + \]  
  \[ E \rightarrow E I + E, + \]
  
  \( \Rightarrow^{E} \)

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

Using Precedence to Solve S/R Conflicts

- Same grammar as before
  
  \[ E \rightarrow E + E \mid E \times E \mid \text{int} \]

- We will also have the states
  
  \[ E \rightarrow E + I E, + \]
  \[ E \rightarrow E + E I, + \]
  \[ E \rightarrow I E + E, + \]
  \[ E \rightarrow E I + E, + \]
  
  \( \Rightarrow^{E} \)

- 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

Precedence Declarations Revisited

- The term "precedence declaration" is misleading!

  These declarations do not define precedence: they define conflict resolutions

  I.e., they instruct shift-reduce parsers to resolve conflicts in certain ways

  These two are not quite the same!
Reduce/Reduce Conflicts

- If a DFA state contains both $[X \rightarrow \alpha I, a]$ and $[Y \rightarrow \beta I, a]$ - 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
  
  $S \rightarrow \varepsilon \mid id \mid id S$

- 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
  
  $[S \rightarrow id I, \ ]$
  $[S' \rightarrow I S, \ ]$
  $[S \rightarrow I, \ ] \Rightarrow^i d [S \rightarrow I, \ ]$
  $[S \rightarrow I id, \ ]$
  $[S \rightarrow I id S, \ ]$

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

- Better to rewrite the grammar as: $S \rightarrow \varepsilon \mid id S$

Using Parser Generators

- Parser generators automatically 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.

\[
\begin{align*}
E & \rightarrow \text{int } I, \$, + \\
E & \rightarrow \text{int } I, \$+/+
\end{align*}
\]

- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core

- We obtain

\[
\begin{align*}
E & \rightarrow \text{int } I, \$, +/+
\end{align*}
\]

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

\[
\{[X \rightarrow \alpha I \beta, b], [Y \rightarrow \gamma I \delta, d]\}
\]

is

\[
\{X \rightarrow \alpha I \beta, Y \rightarrow \gamma I \delta\}
\]

LALR States

- Consider for example the LR(1) states

\[
\begin{align*}
\{[X \rightarrow \alpha I, a], [Y \rightarrow \beta I, c]\} \\
\{[X \rightarrow \alpha I, b], [Y \rightarrow \beta I, d]\}
\end{align*}
\]

- They have the same core and can be merged

- The merged state contains:

\[
\{[X \rightarrow \alpha I, a/b], [Y \rightarrow \beta I, c/d]\}
\]

- 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
  \[[X \rightarrow \alpha 1, a], [Y \rightarrow \beta 1, b]\]
  \[[X \rightarrow \alpha 1, b], [Y \rightarrow \beta 1, a]\]
- And the merged LALR(1) state
  \[[X \rightarrow \alpha 1, a/b], [Y \rightarrow \beta 1, a/b]\]
- Has a new reduce/reduce conflict
- In practice such cases are rare

LALR vs. LR Parsing: Things to keep in mind

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

A Hierarchy of Grammar Classes

From Andrew Appel, "Modern Compiler Implementation in ML"