LR Parsing
LALR Parser Generators
Outline

• 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
  \[ \alpha \vdash \gamma \]
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is the string of terminals not yet examined
• Initially: \( \text{I } x_1 x_2 \ldots x_n \)
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
  
  $$E + (\text{int}) \Rightarrow E + (\text{int})$$

• **Reduce** pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)
  
  $$E + (E + (E) \mid \text{int}) \Rightarrow E + (E \mid \text{int})$$
**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 "$A \rightarrow \beta$ on tok" then **reduce**
LR(1) Parsing: An Example

1. int + (int) + (int)$ shift
2. int I + (int) + (int)$ E → int
3. E + (int) + (int)$ shift (x3)
4. E + (E I) + (int)$ E → int
5. E + (E I) + (int)$ shift
6. E + (E I) + (int)$ E → E+(E)
7. E + (int)$ shift (x3)
8. E + (int I)$ E → int
9. E + (E I)$ shift
10. E + (E I)$ E → E+(E)
11. E $ accept

E → int
E → E + int
E → int + (int)
E → E + (E)
E → (E) + (int)
E → E + (E)
E → int$
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
Representing the DFA: Example

The table for a fragment of our DFA:

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

sk is shift and goto state k
rX→α is reduce
gk 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
  \[ \langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle \]
  \text{state}_k \text{ is the final state of the DFA on sym}_1 \ldots \text{sym}_k
The LR Parsing Algorithm

let $I = w\$ be initial input
let $j = 0$
let DFA state 0 be the start state
let stack $= \langle \text{dummy}, 0 \rangle$
repeat
    case action[top_state(stack), $I[j]$] of
    shift $k$: push $\langle I[j++], k \rangle$
    reduce $X \rightarrow A$:
    pop $|A|$ pairs,
    push $\langle X, \text{goto}[\text{top_state(stack)}, X] \rangle$
    accept: halt normally
    error: halt and report error
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 $\text{int}$ or an $E + (E)$ RHS
LR(0) Items

• An LR(0) item is a production with a “$\epsilon$” somewhere on the RHS

• The LR(0) items for $T \rightarrow (E)$ are
  - $T \rightarrow \epsilon \ (E)$
  - $T \rightarrow (\epsilon \ E)$
  - $T \rightarrow (E \ \epsilon)$
  - $T \rightarrow (E) \ \epsilon$

• The only LR(0) item for $X \rightarrow \epsilon$ is $X \rightarrow \epsilon$
LR(0) Items: Intuition

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

- Notes:
  - \([X \rightarrow \alpha \mid a\beta]\) means that \(a\) should follow
    • Then we can shift it and still have a viable prefix
  - \([X \rightarrow \alpha \mid \_]\) means that we could reduce \(X\)
    • But this is not always a good idea!
LR(1) Items

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

• \([X \rightarrow \alpha \mid \beta, a]\) describes a context of the parser
  
  - We are trying to find an \( X \) followed by an \( a \), and
  
  - We have (at least) \( \alpha \) 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
  - $\alpha I \gamma$, where $\alpha$ is the stack and $\gamma$ is the remaining string of terminals
• In items, $I$ is used to mark a prefix of a production RHS:
  \[ X \rightarrow \alpha I \beta, \alpha \]
  - Here $\beta$ 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 $S \rightarrow E$
  - Where $E$ is the old start symbol

- The initial parsing context contains:
  
  $S \rightarrow \epsilon 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
    \[ E \rightarrow E + (\epsilon E), + \]

- In context containing
  \[ E \rightarrow E + (E), + \]
  - We can perform a reduction with \( E \rightarrow E + (E) \)
  - But only if a + follows
LR(1) Items (Cont.)

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

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

\[
\text{Closure}(\text{Items}) =
\begin{align*}
\text{repeat} & \\
& \text{for each } [X \rightarrow \alpha \mid \beta, a] \text{ in Items} \\
& \text{for each production } Y \rightarrow \gamma \\
& \text{for each } b \text{ in First}(\beta a) \\
& \text{add } [Y \rightarrow \gamma, b] \text{ to Items} \\
& \text{until Items is unchanged}
\end{align*}
\]
Constructing the Parsing DFA (1)

• Construct the start context:

\[ \text{Closure(}\{S \rightarrow I E, \$\}\) \]

\[
S \rightarrow I E , \$
E \rightarrow I E+(E) , \$
E \rightarrow I \text{int} , \$
E \rightarrow I E+(E) , +
E \rightarrow I \text{int} , +
\]

• We abbreviate as:

\[
S \rightarrow I E , \$
E \rightarrow I E+(E) , \$/+
E \rightarrow I \text{int} , \$/+
\]
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 \([X \rightarrow a I , b]\) is labeled with "reduce with \(X \rightarrow a\) on \(b\)"

• And now the transitions ...
The DFA Transitions

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

```
Transition(State, y)
Items = \emptyset
for each $[X \rightarrow \alpha \mid y\beta, b]$ in State
  add $[X \rightarrow \alpha y \mid \beta, b]$ to Items
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
  \[X \rightarrow \alpha \mid a\beta, b\] and \[Y \rightarrow \gamma \mid, a\]

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

• This is called a shift-reduce conflict
Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  \[ S \rightarrow \text{if E then S} \mid \text{if E then S else S} \mid \text{OTHER} \]
- Will have a DFA state containing
  \[
  \begin{align*}
  [S &\rightarrow \text{if E then S I, else}] \\
  [S &\rightarrow \text{if E then S I else S, x}]
  \end{align*}
  \]
- 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

• Consider the ambiguous grammar
  \[ E \rightarrow E + E \mid E * E \mid \text{int} \]

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

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

\[ [E \rightarrow E * I E, +] \quad [E \rightarrow E * E I, +] \]
\[ [E \rightarrow I E + E, +] \Rightarrow^E [E \rightarrow E I + E, +] \]

\[ \ldots \]

\[ \ldots \]

• 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
  \[ E \rightarrow E + E \mid E * E \mid \text{int} \]

• We will also have the states
  \[ [E \rightarrow E + I \ E, \ +] \quad [E \rightarrow E + E \ I, \ +] \]
  \[ [E \rightarrow I \ E + E, \ +] \quad \Rightarrow^E \quad [E \rightarrow E \ I + E, \ +] \]
  \[ \ldots \quad \ldots \]

• 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
Using Precedence to Solve S/R Conflicts

• Back to our dangling else example
  
  \[ S \rightarrow \text{if } E \text{ then } S \mid, \quad \text{else} \]
  
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{else } S, \quad x \]

• Can eliminate conflict by declaring \text{else} having higher precedence than \text{then}

• But this starts to look like “hacking the tables”

• Best to avoid overuse of precedence declarations or we will end with unexpected parse trees
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
  \[ S \rightarrow \varepsilon \mid \text{id} \mid \text{id} \ S \]

• There are two parse trees for the string \text{id}
  \[ S \rightarrow \text{id} \]
  \[ S \rightarrow \text{id} \ S \rightarrow \text{id} \]

• How does this confuse the parser?
More on Reduce/Reduce Conflicts

• Consider the states

\[
\begin{align*}
[S' \rightarrow S, \ \$] & \quad [S \rightarrow \text{id} \ S, \ \$] \\
[S \rightarrow \text{id} \ I, \ \$] & \quad [S \rightarrow \text{id} \ S, \ \$] \\
[S \rightarrow \text{id} \ S, \ \$] & \quad [S \rightarrow \text{id} \ S, \ \$]
\end{align*}
\]

⇒ \text{id}

• Reduce/reduce conflict on input $\

S' \rightarrow S \rightarrow \text{id}

S' \rightarrow S \rightarrow \text{id} \ S \rightarrow \text{id}

• Better to rewrite the grammar as: 

\[S \rightarrow \varepsilon \mid \text{id} \ S\]
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.

  \[
  \begin{align*}
  E \rightarrow \text{int } 1, \$/+ & \quad 1' \\
  E \rightarrow \text{int } & \quad 5
  \end{align*}
  \]

  and

  \[
  \begin{align*}
  E \rightarrow \text{int } 1, )/+ & \\
  E \rightarrow \text{int } & \\
  \end{align*}
  \]

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

• We obtain
**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

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

is

\[
\{X \rightarrow \alpha \mid \beta, Y \rightarrow \gamma \mid \delta\}
\]
LALR States

- Consider for example the LR(1) states
  
  \{[X \rightarrow \alpha \downarrow, a], [Y \rightarrow \beta \downarrow, c]\}
  
  \{[X \rightarrow \alpha \downarrow, b], [Y \rightarrow \beta \downarrow, d]\}

- They have the same core and can be merged

- The merged state contains:
  
  \{[X \rightarrow \alpha \downarrow, a/b], [Y \rightarrow \beta \downarrow, 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.

Diagram showing the LR(1) and LALR(1) parsing tables and transitions.

LR(1) Table:
E → int on $, +
E → int on $, +, )
E → E + (E) on $, +
E → E + (E) on $, +, )
accept on $
The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states
  \[
  \{ [X \rightarrow \alpha \ I \ a], [Y \rightarrow \beta \ I \ b] \} \\
  \{ [X \rightarrow \alpha \ I \ b], [Y \rightarrow \beta \ I \ a] \} \\
  \]
- And the merged LALR(1) state
  \[
  \{ [X \rightarrow \alpha \ I \ a/b], [Y \rightarrow \beta \ I \ a/b] \} \\
  \]
- 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"