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
  \[ \alpha \mid \gamma \]
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is the string of terminals not yet examined

• Initially: \( l \ x_1 x_2 \ldots x_n \)
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 off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)
    \[ E + (E + (E) \text{int}) \Rightarrow E + (E \text{int}) \]
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

```
int 1 + (int) + (int)$  shift
int 1 + (int) + (int)$  E → int
E 1 + (int) + (int)$  shift (x3)
E 1 + (int)$  E → int
t + E 1 + (int)$  E → int
t E 1 + (int)$  shift
t E 1 $  E → E+(E)
t E$  accept
```
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>int</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r_E \rightarrow \text{int}</td>
<td>r_E \rightarrow \text{int}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r_E \rightarrow E+(E)</td>
<td>r_E \rightarrow E+(E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **sk** is shift and goto state k
- **r\_X \rightarrow \alpha** 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

- Remember for each stack element on which state it brings the DFA

- LR parser maintains a stack
  \[ \langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle \]
  \( \text{state}_k \) is the final state of the DFA on \( \text{sym}_1 \ldots \text{sym}_k \)
The LR Parsing Algorithm

let I = w$ be initial input
let j = 0
let DFA state 0 be the start state
let stack = ⟨ dummy, 0 ⟩
  repeat
    case action[top_state(stack), I[j]] of
      shift k:  push ⟨ I[j++], k ⟩
      reduce X → A:
        pop |A| pairs,
        push ⟨ X, goto[top_state(stack), X] ⟩
      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 int or an $E + (E)$ RHS
LR(0) Items

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

• The items for $T \rightarrow (E)$ are
  $T \rightarrow I \ (E)$
  $T \rightarrow (I \ E)$
  $T \rightarrow (E \ I)$
  $T \rightarrow (E) \ I$

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

• An item \([X \rightarrow \alpha \mid \beta]\) says that
  - the parser is looking for an \(X\)
  - it 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
  - \([X \rightarrow \alpha \mid \lambda]\) means that we could reduce \(X\)
    • But this is not always a good idea!
LR(1) Items

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

• \([X \rightarrow \alpha I \beta, \ a]\) 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:
  \[ X \rightarrow \alpha I \beta, \quad \alpha \]
  - Here $\beta$ might contain terminals as well

• In both 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:
  
  $S \rightarrow I \ E \ , \ \$$
  
  - Trying to find an $S$ as a string derived from $E$$
  - The stack is empty
LR(1) Items (Cont.)

- In context containing
  \[ E \rightarrow E + (E), + \]
  - 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
LR(1) Items (Cont.)

- Consider the item
  \[ E \rightarrow E + ( I E ) , + \]
- We expect a string derived from \( E ) + \)
- There are 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:
  \[ E \rightarrow I \text{int} , ) \]
  \[ E \rightarrow I E + ( E ) , ) \]
The Closure Operation

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

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

• Construct the start context: Closure({S → I E, $})

  S → I E , $
  E → I E+(E), $
  E → I int , $
  E → I E+(E), +
  E → I int , +

• We abbreviate as:

  S → I E , $
  E → I E+(E) , $/+
Constructing the Parsing DFA (2)

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

- The start state contains $[S \rightarrow l\ E\ ,\ \$]$

- A state that contains $[X \rightarrow \alpha\ l,\ b]$ is labelled with “reduce with $X \rightarrow \alpha$ 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 "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

S → 1 E , $
E → 1 E+(E), $/+ 
E → 1 int , $/+ 

E → int 1, $/+ 
E → int on $, + 
E → E+ 1 (E), $/+ 
E → E+(E), $/+ 
E → int 1(1), $/+ 
E → int 1 on $, + 
E → int on $, + 

and so on...

Compiler Design I (2011)
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

\[
S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER}
\]

• Will have DFA state containing

\[
[S \rightarrow \text{if } E \text{ then } S, \quad \text{else}]
\]
\[
[S \rightarrow \text{if } E \text{ then } S \text{ else } S, \quad x]
\]

• If else follows then we can shift or reduce

• Default (\textit{yacc, ML-yacc, 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:
  \[
  \%\text{left} + \\
  \%\text{left} *
  \]

- 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:

\[
\begin{align*}
[E &\rightarrow E \cdot I E, +] &\quad [E \rightarrow E \cdot E I, +] \\
[E &\rightarrow I E + E, +] &\Rightarrow^E [E \rightarrow E I + E, +]
\end{align*}
\]

\[\ldots \quad \ldots\]

• Will choose reduce because precedence of rule \( E \rightarrow E \cdot E \) is higher than of terminal +
Using Precedence to Solve S/R Conflicts

• Same grammar as before
  \[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]

• We will also have the states
  \[
  \begin{align*}
  [E \rightarrow E + I \ E, +] & \quad [E \rightarrow E + E I, +] \\
  [E \rightarrow I E + E, +] \Rightarrow^E [E \rightarrow E I + E, +] \\
  \ldots & \quad \ldots
  \end{align*}
  \]

• 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
  \[S \rightarrow \text{if } E \text{ then } S \text{ I, else}\]
  \[S \rightarrow \text{if } E \text{ then } S \text{ I else } S, \ x]\n- Can eliminate conflict by declaring else having higher precedence than 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
The two are not quite the same thing!
Reduce/Reduce Conflicts

• If a DFA state contains both 
  
  \[ X \rightarrow \alpha \, \text{I, a} \] and \[ Y \rightarrow \beta \, \text{I, a} \]
  
  - 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**
  
  \[
  \begin{align*}
  &\text{[S → id I,}$ $] \\
  &\text{[S' → I S,}$ $] \\
  &\text{[S → I,}$ $] \ ⇒^\text{id} \ [S → I,}$ $] \\
  &\text{[S → I id,}$ $] \\
  &\text{[S → I id S,}$ $] \\
  &\text{[S → id I S,}$ $] \\
  &\text{[S → id S,}$ $] \\
  \end{align*}
  \]

- **Reduce/reduce conflict on input $**
  
  \[
  \begin{align*}
  &S' \rightarrow S \rightarrow \text{id} \\
  &S' \rightarrow S \rightarrow \text{id S} \rightarrow \text{id}
  \end{align*}
  \]

- **Better rewrite the grammar:**
  
  \[
  S \rightarrow \varepsilon \mid \text{id S}
  \]
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.

\[
\begin{align*}
\text{1} & : & \text{E} & \to \text{int I, } $/+ \\
\text{4} & : & \text{E} & \to \text{int I, } $/+ \\
\text{5} & : & \text{E} & \to \text{int I, } )/+ \\
\text{6} & : & \text{E} & \to \text{int I, } )/+ \\
\end{align*}
\]

and

\[
\begin{align*}
\text{1'} & : & \text{E} & \to \text{int I, } $/+ />\\
\end{align*}
\]

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

• We obtain
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
  \[
  \{[X \rightarrow \alpha \, 1, \, a], \, [Y \rightarrow \beta \, 1, \, c]\}
  \{[X \rightarrow \alpha \, 1, \, b], \, [Y \rightarrow \beta \, 1, \, d]\}
  \]
- They have the same core and can be merged
- And the merged state contains:
  \[
  \{[X \rightarrow \alpha \, 1, \, a/b], \, [Y \rightarrow \beta \, 1, \, c/d]\}
  \]
- These are called **LALR(1)** states
  - Stands for Look Ahead 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

A → B → C
D → E → F

A → B → E → C
D → E → F
Conversion LR(1) to LALR(1): Example.
The LALR Parser Can Have Conflicts

• Consider for example the LR(1) states
  \{[X \to \alpha I, a], [Y \to \beta I, b]\}
  \{[X \to \alpha I, b], [Y \to \beta I, a]\}
• And the merged LALR(1) state
  \{[X \to \alpha I, a/b], [Y \to \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 for parser generators
A Hierarchy of Grammar Classes

From Andrew Appel, "Modern Compiler Implementation in ML"
Semantic Actions in LR Parsing

• We can now illustrate how semantic actions are implemented for LR parsing
• Keep attributes on the stack

• On shifting $a$, push attribute for $a$ on stack
• On reduce $X \rightarrow \alpha$
  - pop attributes for $\alpha$
  - compute attribute for $X$
  - and push it on the stack
Performing Semantic Actions: Example

• Recall the example

\[
E \rightarrow T + E_1 \quad \{ \text{E.val} = \text{T.val} + \text{E}_1\text{.val} \}
\]

\[
| \quad T \quad \{ \text{E.val} = \text{T.val} \}
\]

\[
T \rightarrow \text{int} \ast T_1 \quad \{ \text{T.val} = \text{int.val} \ast \text{T}_1\text{.val} \}
\]

\[
| \quad \text{int} \quad \{ \text{T.val} = \text{int.val} \}
\]

• Consider the parsing of the string \(3 \ast 5 + 8\)
Performing Semantic Actions: Example

<table>
<thead>
<tr>
<th>int * int + int</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>int$_3$ * int + int</td>
<td></td>
</tr>
<tr>
<td>int$_3$ * int + int</td>
<td></td>
</tr>
<tr>
<td>int$_3$ * int$_5$ + int</td>
<td>reduce T → int</td>
</tr>
<tr>
<td>int$_3$ * T$_5$ + int</td>
<td></td>
</tr>
<tr>
<td>T$_{15}$ + int</td>
<td></td>
</tr>
<tr>
<td>T$_{15}$ + int</td>
<td></td>
</tr>
<tr>
<td>T$_{15}$ + int$_8$</td>
<td></td>
</tr>
<tr>
<td>T$_{15}$ + T$_8$</td>
<td></td>
</tr>
<tr>
<td>T$_{15}$ + E$_8$</td>
<td></td>
</tr>
<tr>
<td>E$_{23}$</td>
<td>accept</td>
</tr>
</tbody>
</table>

3 * 5 + 8
Notes

• The previous example shows how synthesized attributes are computed by LR parsers

• It is also possible to compute inherited attributes in an LR parser
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)
  - LALR(1) parser generators

• Next time we move on to semantic analysis
Supplement to LR Parsing

Strange Reduce/Reduce Conflicts due to LALR Conversion (and how to handle them)
Strange Reduce/Reduce Conflicts

• Consider the grammar

\[
\begin{align*}
S & \rightarrow P \ R , \\
NL & \rightarrow N \mid N , NL \\
P & \rightarrow T \mid NL : T \\
R & \rightarrow T \mid N : T \\
N & \rightarrow id \\
T & \rightarrow id
\end{align*}
\]

• \(P\) - parameters specification
• \(R\) - result specification
• \(N\) - a parameter or result name
• \(T\) - a type name
• \(NL\) - a list of names
Strange Reduce/Reduce Conflicts

• In $P$ an id is a
  - $N$ when followed by $,$ or $:$
  - $T$ when followed by id

• In $R$ an id is a
  - $N$ when followed by $:$
  - $T$ when followed by $,$

• This is an LR(1) grammar

• But it is not LALR(1). Why?
  - For obscure reasons
A Few LR(1) States

1. \( P \rightarrow \text{I T id} \)
2. \( R \rightarrow \text{I T} , \)
3. \( T \rightarrow \text{id I id} \)
4. \( T \rightarrow \text{id I id} , \)
   \( N \rightarrow \text{id I} , \)
   \( N \rightarrow \text{id I} : \)

LALR reduce/reduce conflict on “,“
What Happened?

- Two distinct states were confused because they have the same core
- Fix: add dummy productions to distinguish the two confused states
- E.g., add

  \[ R \rightarrow \text{id} \text{ bogus} \]

  - \text{bogus} is a terminal not used by the lexer
  - This production will never be used during parsing
  - But it distinguishes \( R \) from \( P \)
A Few LR(1) States After Fix

Different cores ⇒ no LALR merging