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: $\text{I} x_1 x_2 \ldots x_n$

The Shift and Reduce Actions (Review)
• Recall the CFG: $E \rightarrow \text{int} \mid E + (E)$
• 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)
    
    $E + (E + (E)) \Rightarrow E + (E)$
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

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>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(E \rightarrow \text{int})</td>
<td>(r_{E \rightarrow \text{int}})</td>
<td>(r_{E \rightarrow \text{int}})</td>
<td></td>
<td>(g_6)</td>
</tr>
<tr>
<td>6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(E \rightarrow E + (E)) on $, +</td>
<td>(r_{E \rightarrow E+(E)})</td>
<td>(r_{E \rightarrow E+(E)})</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 \text{ is the final state of the DFA on } \text{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 items for \( T \to (E) \) are
  \[ T \to \text{i} (E) \]
  \[ T \to (\text{i} E) \]
  \[ T \to (E \text{i}) \]
  \[ T \to (E) \text{i} \]
- The only item for \( X \to \varepsilon \) is \( X \to \text{i} \)
LR(0) Items: Intuition

- An item $[X \rightarrow \alpha \ I \beta]$ 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:
  - $[X \rightarrow \alpha \ I \alpha \beta]$ means that $\alpha$ should follow. Then we can shift it and still have a viable prefix
  - $[X \rightarrow \alpha \ I \beta]$ means that we could reduce $X$
    - But this is not always a good idea!

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:
  - $X \rightarrow \alpha \ I \beta$, $a$
    - Here $\beta$ might contain terminals as well
- In both case the stack is on the left of $I$

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 $\beta a$

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

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

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

Constructing the Parsing DFA (1)

- Construct the start context:

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

- We abbreviate as:

\[
\begin{align*}
S & \rightarrow i E , \$
E & \rightarrow i E+(E) , \$/+
E & \rightarrow i \text{int} , \$/+
\end{align*}
\]
Constructing the Parsing DFA (2)

- A DFA state is a closed set of LR(1) items
- The start state contains \([S \rightarrow \varepsilon E, \$]\)
- A state that contains \([X \rightarrow \alpha I_y \beta, b]\) is labelled with “reduce with \(X \rightarrow \alpha\) on \(b\)”
- And now the transitions …

The DFA Transitions

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

\[
\text{Transition}(\text{State}, y) \quad \text{Items} = \emptyset
\]

for each \([X \rightarrow \alpha I_y \beta, b]\) in State

add \([X \rightarrow \alpha y I \beta, b]\) to Items

return \(\text{Closure}()\)

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 \([X \rightarrow \alpha a I \beta, b]\), 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
  \([S \rightarrow \text{if } E \text{ then } S \mid \text{else}]\)
  \([S \rightarrow \text{if } E \text{ then } S \text{ else } S, x]\)

- If else follows then we can shift or reduce

- Default (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 \quad \([E \rightarrow E I + E, +]\)

- 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:

  \[ E \rightarrow E \ast I E, + \]  \[ E \rightarrow E \ast E I, + \]

  \[ E \rightarrow I E + E, + \]  \[ E \rightarrow E I + E, + \]

  \[ E \rightarrow E \ast E I, + \]

  \[ E \rightarrow E + E I, + \]

  \[ E \rightarrow E + E \]

  \[ E \rightarrow I E + E, + \]

  \[ E \rightarrow E I + E, + \]

  \[ E \rightarrow E \ast I E, + \]

  \[ E \rightarrow E \ast E I, + \]

  \[ E \rightarrow I E + E, + \]

  \[ E \rightarrow I E + E, + \]  \[ E \rightarrow E I + E, + \]

  \[ E \rightarrow E I + E, + \]  \[ E \rightarrow E I + E, + \]

  \[ E \rightarrow E \ast E I, + \]  \[ E \rightarrow E + E I, + \]

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

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 
  \[X \rightarrow \alpha I, a]\] 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

More on Reduce/Reduce Conflicts

• Consider the states 
  \([S \rightarrow \text{id} I, ~] \) 
  \([S' \rightarrow I S, ~] \) 
  \([S \rightarrow I, ~] \Rightarrow_{id} [S \rightarrow I, ~] \) 
  \([S \rightarrow I \text{id}, ~] \) 
  \([S \rightarrow I \text{id} S, ~] \) 
• Reduce/reduce conflict on input $ 
  S' \rightarrow S \rightarrow \text{id} 
  S' \rightarrow S \rightarrow \text{id} S \rightarrow \text{id} 
• Better 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

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 \( i d \)
  
  \( S \rightarrow \text{id} \)
  \( S \rightarrow \text{id} S \rightarrow \text{id} \)
• How does this confuse the parser?
LR(1) Parsing Tables are Big

• But many states are similar, e.g.

\[
E \rightarrow \text{int } I \, $/+
\]

and

\[
E \rightarrow \text{int } I \, )/+\]

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

• We obtain

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

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

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

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

• They have the same core and can be merged
  - And 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
  \[\{[X \rightarrow \alpha I, a], [Y \rightarrow \beta I, b]\}\]
  \[\{[X \rightarrow \alpha I, b], [Y \rightarrow \beta I, a]\}\]
- 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 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

$$E \rightarrow T \cdot E_1 \quad \{ E.val = T.val + E_1.val \}$$
$$\mid T \quad \{ E.val = T.val \}$$
$$T \rightarrow int \cdot T_1 \quad \{ T.val = int.val \cdot T_1.val \}$$
$$\mid int \quad \{ T.val = int.val \}$$

Consider the parsing of the string: $4 \cdot 9 + 6$

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

Strange Reduce/Reduce Conflicts

- Consider the grammar
  
  \[
  S \rightarrow P \ R \ , \quad NL \rightarrow N \ | \ N \ , \ NL \\
  P \rightarrow T \ | \ NL \ : \ T \\
  R \rightarrow T \ | \ N \ : \ T \\
  N \rightarrow \text{id} \\
  T \rightarrow \text{id}
  \]

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

Supplement to LR Parsing

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

- 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

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

R → id bogus,