Conflicts in LR Parsing and More LR Parsing Types

Lecture 10

Dr. Sean Peisert – ECS 142 – Spring 2009
Status

• Project 2 Due Friday, Apr. 24, 11:55pm
• The usual lecture time is being replaced by a discussion section on Friday, too.
• SemCheck
LR Parsing Tables

• 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:
  \([X \rightarrow \alpha \cdot a\beta, b]\) and \([Y \rightarrow \gamma \cdot, a]\)

• Then on input “a” we could either:
  • Shift into state \([X \rightarrow \alpha a \cdot \beta, b]\), 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 a DFA state containing
  \[ [S \rightarrow \text{if } E \text{ then } S\star, \quad \text{else}] \]
  \[ [S \rightarrow \text{if } E \text{ then } S\star \text{ else } S, \quad \times ] \]
- If “else” follows then we can shift or reduce
- Default (bison, CUP, etc..) is to shift.
  - Default behavior is as needed in this case.
Shift/Reduce Conflicts

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

• Will have the DFA states containing
  \[ [E \rightarrow E \ast \cdot E, +] \quad [E \rightarrow E \ast E\cdot, +] \]
  \[ [E \rightarrow \cdot E + E, +] \quad \Rightarrow^E \quad [E \rightarrow E\cdot + E, +] \]

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

• Example: 1 - 2 - 5

• Stack contains “1-2” and the look-ahead is “-”

• Left associative: (1 - 2) - 5

• Right associative: 1 - (2 - 5)
Shift/Reduce Conflicts

- In bison declare precedence and associativity:
  \%left + (left associative)
  \%left * (left associative, higher precedence)
- Precedence of a rule = that of its last terminal
  - See bison manual ("Conflict-Dependent Precedence") for ways to override this default
- Bison resolves shift/reduce conflicts 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

- Example:
  \[ E \rightarrow E \ast \cdot E, + \]  \[ E \rightarrow E \ast E\cdot, + \]
  \[ E \rightarrow \cdot E + E, + \]  \[ E \rightarrow E\cdot + E, + \]

- Will choose reduce because precedence of rule \( E \rightarrow E \ast E \) is higher than of terminal +
- (Last terminal in \( E \rightarrow E \ast E \) is “\( \ast \)”)
Using Precedence to Solve S/R Conflicts

• Grammar:
  \[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]

• States:
  \[ [E \rightarrow E + \cdot E, +] \quad \Rightarrow^E \quad [E \rightarrow E \cdot + E, +] \]

• Now we also have a shift/reduce on input +
  • Reduce, because \( E \rightarrow E + E \) and \( + \) have the same precedence and \( + \) is left-associative.
Using Precedence to Solve S/R Conflicts

- Dangling else
  \[S \rightarrow \text{if } E \text{ then } S^\bullet \text{, else}\]
  \[S \rightarrow \text{if } E \text{ then } S^\bullet \text{ else } S^\bullet \text{, } x\]

- Can eliminate conflict by declaring “else” with higher precedence than “then”
- Or just rely on the default shift action
- But this starts to look like “hacking the parser”
- Best to avoid overuse of precedence declarations or you’ll end up with unexpected parse trees.
Rewriting Grammar to Solve S/R Conflicts

• Ambiguous grammar (w/o precedence):
  \[ E \rightarrow E + E | E \ast E | (E) | \text{int} \]

• Input:
  \[ \text{int} + \text{int} \ast \text{int} \]

• Rewritten Grammar:
  \[ E \rightarrow E + T | T \quad \text{gives } + \text{ a lower precedence} \]
  \[ T \rightarrow T \ast F | F \quad \text{gives } (, ) \text{ a higher precedence} \]
  \[ F \rightarrow (E) | \text{int} \quad \text{and only ONE parse tree} \]
Reduce/Reduce Conflicts

- If a DFA state contains both:
  \[ X \rightarrow \alpha\bullet, a \] and \[ Y \rightarrow \beta\bullet, a \]

- 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?
Reduce/Reduce Conflicts

- Consider states:
  \[ S' \rightarrow \cdot S, \quad $ \]
  \[ S \rightarrow \cdot, \quad $ \]
  \[ S \rightarrow \cdot \text{id}, \quad $ \]
  \[ S \rightarrow \cdot \text{id} S, \quad $ \]

- 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 $ S \rightarrow \varepsilon \mid \text{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.:
  \[(1) \ E \rightarrow \text{int}^*, \$/+ \quad \text{red.} \ E \rightarrow \text{int on} \ $, +\]
  and
  \[(5) \ E \rightarrow \text{int}^*, )/+ \quad \text{red.} \ E \rightarrow \text{int on} \ ), +\]
- Idea: merge the DFA states whose items differ only in the lookahead tokens. Such states have the same “core.” We obtain:
  \[(1') \ E \rightarrow \text{int}^*, \$/+/) \quad \text{red.} \ E \rightarrow \text{int on} \ $, +, )\]
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 

  
  \[ [X \rightarrow \alpha \cdot \beta, b] \text{ and } [Y \rightarrow \gamma \cdot \delta, d] \]

• is

  
  \[ [X \rightarrow \alpha \cdot \beta] \text{ and } [Y \rightarrow \gamma \cdot \delta] \]
LALR States

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

• Same core, different lookahead. They can be merged. Merged state contains:
  \{[X \rightarrow \alpha \cdot, a/b], [Y \rightarrow \beta \cdot, c/d]\}

• These are called LALR(1) states
  • Stands for LookAhead LR
  • Typically 10 times fewer LALR(1) states than LR(1)
LALR(1) DFA

- Repeat until all states have distinct core:
  - Choose two distinct states with the 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.
LR(1) Parsing Automaton for
grammar \( E \rightarrow E + (E) \mid \text{int} \)

LALR(1) Parsing Automaton for
grammar \( E \rightarrow E + (E) \mid \text{int} \)
LALR Parser Can Have Conflicts

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

• Has a new reduce-reduce conflict
• In practice such cases are rare.
• However, no new shift/reduce conflicts. Why?
LALR vs. Canonical LR Parsing

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

• LALR uses “lookahead” sets computed by using “closure” operation.

• SLR only uses “follow” sets.

• “Follow” sets do not have context information, and so create more conflicts.

• LALR can be more selective because it has contexts, and so can avoid conflicts.
Grammar Hierarchy

Re-created from Andrew Appel, *Modern Compiler Implementation in Java*
End Notes

- Parsing:
  - A solid foundation: CFGs
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack on LR(0): LALR(1)
  - LALR(1) Parser Generators: yacc, bison, CUP
- Next Class: Top-Down Parsing — LL(1)