In One Slide

- An LR(1) parsing table can be constructed automatically from a CFG. An LR(1) item is a pair made up of a production and a lookahead token; it represents a possible parser context. After we extend LR(1) items by closing them they become LR(1) DFA states. Grammars can have shift/reduce or reduce/reduce conflicts. You can fix most conflicts with precedence and associativity declarations. LALR(1) tables are formed from LR(1) tables by merging states with similar cores.

Outline

- Review of bottom-up parsing

- Computing the parsing DFA
  - Closures, LR(1) Items, States
  - Transitions

- Using parser generators
  - Handling Conflicts
**Bottom-up Parsing (Review)**

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

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

**Key Issue: When to Shift or Reduce?**

- Idea: use a 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 $\text{tok}$ after $\triangleright$
  - If $X$ has a transition labeled $\text{tok}$ then shift
  - If $X$ is labeled with “$A \rightarrow \beta$ on $\text{tok}$” then reduce
LR(1) Parsing. An Example

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
Parsing Contexts

• Consider the state:

\[
\begin{array}{c}
E \\
\text{int} + ( \text{int} ) + [ \text{int} ]
\end{array}
\]

- The stack is

\[
E + ( \uparrow \text{int} ) + ( \text{int} )
\]

• Context:

- We are looking for an \( E \rightarrow E + ( \bullet E ) \)
  - Have seen \( E + \) from the right-hand side
- We are also looking for \( E \rightarrow \bullet \text{int} \) or \( E \rightarrow \bullet E + ( E ) \)
  - Have seen nothing from the right-hand side
- One DFA state describes several contexts

---

LR(1) Items

• An LR(1) item is a pair:

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

- \( X \rightarrow \alpha \beta \) is a production
- \( a \) is a terminal (the lookahead terminal)
- LR(1) means 1 lookahead terminal

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

describes a context of the parser

- We are trying to find an \( X \) followed by an \( a \), and
- We have \( \alpha \) already on top of the stack
- Thus we need to see next a prefix derived from \( \beta a \)

---

Note

• The symbol \( \uparrow \) was used before to separate the stack from the rest of input
  - \( \alpha \uparrow \gamma \), where \( \alpha \) is the stack and \( \gamma \) is the remaining string of terminals

• In LR(1) items \( \bullet \) is used to mark a prefix of a production rhs:

\[
X \rightarrow \alpha \bullet \beta, a
\]

- Here \( \beta \) might contain non-terminals as well
• In both case the stack is on the left
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
  - No need to do this if $E$ had only one production

- The initial parsing context contains:
  $S \rightarrow \bullet E, S$
  - Trying to find an $S$ as a string derived from $ES$
  - 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 $E \rightarrow E + ( \bullet E ), +$

- In context containing $E \rightarrow E + ( E ) \bullet, +$
  - We can perform a reduction with $E \rightarrow E + ( E )$
  - But only if a $+$ follows

LR(1) Items (Cont.)

- Consider a context with the item $E \rightarrow E + ( \bullet E ) , +$
- We expect next a string derived from $E ) +$
- There are two productions for $E$
  $E \rightarrow \text{int}$ and $E \rightarrow E + ( E )$
- We describe this by extending the context with two more items:
  $E \rightarrow \bullet \text{int}, )$
  $E \rightarrow \bullet E + ( E ) , )
The Closure Operation

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

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

Constructing the Parsing DFA (1)

- Construct the start context:

\[
\text{Closure}(\{S \rightarrow \cdot E, \ \$\}) =
\]
\[
S \rightarrow \cdot E, \ $
E \rightarrow \cdot E+(E), \ $
E \rightarrow \cdot \text{int}, \ $
E \rightarrow \cdot E+(E), \ +
E \rightarrow \cdot \text{int}, \ +
\]

- We abbreviate as:

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

Constructing the Parsing DFA (2)

- An LR(1) DFA state is a closed set of LR(1) items.
  - This means that we performed Closure.

- The start state contains \([S \rightarrow \cdot E, \ \$]\).

- A state that contains \([X \rightarrow \alpha\epsilon, \ b]\) is labeled with “reduce with \(X \rightarrow \alpha\) on \(b\)”.

- And now the transitions …
The DFA Transitions

- A state “State” that contains $[X \rightarrow \alpha y\beta, b]$ 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 $\leftarrow \emptyset$
for each $[X \rightarrow \alpha y\beta, b] \in$ State
  add $[X \rightarrow \alpha y\beta, b]$ to Items
return Closure(Items)

LR(1) DFA Construction Example

LR Parsing Tables. Notes

- Parsing tables (= the DFA) can be constructed automatically for a CFG
  - “The tables which cannot be constructed are constructed automatically in response to a CFG input. You asked for a miracle, Theo. I give you the L-R-1.” - Hans Gruber, Die Hard
- 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 a\beta, b]\) and \([Y \rightarrow \gamma, a]\)

- 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 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
  \[\begin{align*}
  &S \rightarrow \text{if } E \text{ then } S, \text{ else} \\
  &S \rightarrow \text{if } E \text{ then } S, \text{ else } S, \ x
  \end{align*}\]
- If else follows then we can shift or reduce
- Default (bison, CUP, 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
  \[\begin{align*}
  &E \rightarrow E + E, + \\
  &E \rightarrow E * E, + \\
  &E \rightarrow E * E \rightarrow E + E, +
  \end{align*}\]
- Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Solution: declare the precedence of * and +
More Shift/Reduce Conflicts

- In bison declare \texttt{precedence} and \texttt{associativity}:
  \begin{verbatim}
  %left + %left * // high precedence
  \end{verbatim}
- \textbf{Precedence} of a rule = that of its last terminal
  - See bison manual for ways to override this default
- Resolve shift/reduce conflict with a \texttt{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:
  \[
  \begin{align*}
  E & \rightarrow E \ast E, + \\
  E & \rightarrow E \ast E + E, + \\
  E & \rightarrow E + E, + \\
  \Rightarrow & E \\
  \end{align*}
  \]
- Will choose \texttt{reduce} on input + because precedence of rule \( E \rightarrow E \ast E \) is higher than of terminal +

Using Precedence to Solve S/R Conflicts

- Same grammar as before
  \[
  E \rightarrow E + E | E \ast E | \text{int}
  \]
- We will also have the states
  \[
  \begin{align*}
  E & \rightarrow E + E, + \\
  E & \rightarrow E \ast E, + \\
  E & \rightarrow E + E, + \\
  \Rightarrow & E \\
  \end{align*}
  \]
- Now we also have a shift/reduce on input +
  - We choose \texttt{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
  \[ S \rightarrow \text{if } E \text{ then } S \] 
  \[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \] 
- 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”
- Avoid overuse of precedence declarations or you’ll end with unexpected parse trees
  - The kiss of death ...

Reduce/Reduce Conflicts

- If a DFA state contains both
  \[ X \rightarrow \alpha, a \] and \[ Y \rightarrow \beta, a \]
  - 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 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
  \[ S \rightarrow \text{id}, \quad S \]
  \[ S' \rightarrow \text{id}, \quad S \]
  \[ S \rightarrow \text{id} \cdot S, \quad S \]
  \[ S \rightarrow \text{id} \cdot S, \quad S \]
  \[ S \rightarrow \text{id} \cdot \text{id}, \quad S \]
  \[ S \rightarrow \text{id} \cdot \text{id}, \quad S \]
  \[ S \rightarrow \text{id} \cdot \text{id} S, \quad S \]
  \[ S \rightarrow \text{id} \cdot \text{id} S, \quad S \]
- Reduce/reduce conflict on input $\text{id}$

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
  - *Why might that be?*
LR(1) Parsing Tables are Big

- But many states are similar, e.g.
  
  \[
  E \rightarrow \text{int}\star, \$/\star \quad E \rightarrow \text{int} \\
  \text{on } \$, \star \quad \text{and} \quad E \rightarrow \text{int}\star, \gamma/\star \quad E \rightarrow \text{int} \\
  \text{on } \gamma, \star
  \]

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

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

LALR States

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

- They have the same core and can be merged
- And the merged state contains:
  
  \[
  \{ [X \rightarrow \alpha\beta, a/b], [Y \rightarrow \beta\delta, c/d] \}
  \]

- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10x fewer LALR(1) states than LR(1)
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

Example LALR(1) to LR(1)

The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states
  - \([X \rightarrow \alpha a, [Y \rightarrow \beta b, b]]\)
  - \([X \rightarrow \alpha a, [Y \rightarrow \beta b, a]]\)
- And the merged LALR(1) state
  - \([X \rightarrow \alpha a/b, [Y \rightarrow \beta a/b]]\)
- Has a new reduce-reduce conflict
- In practice such cases are rare
LALR vs. LR Parsing

- LALR languages are not natural
  - They are an efficiency hack on LR languages

- Any “reasonable” programming language has a LALR(1) grammar

- LALR(1) has become a standard for programming languages and for parser generators

A Hierarchy of Grammar Classes

From Andrew Appel, “Modern Compiler Implementation in Java”

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

- Now we move on to semantic analysis
Supplement to LR Parsing

Strange Reduce/Reduce Conflicts
Due to LALR Conversion
(from the bison manual)

Strange Reduce/Reduce Conflicts

- Consider the grammar
  
  \[
  S \rightarrow PR, \quad NL \rightarrow N \mid N, NL \\
  P \rightarrow T \mid NL : T \quad R \rightarrow T \mid N : T \\
  N \rightarrow id \quad T \rightarrow id 
  \]

- **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

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 \( \Rightarrow \) no LALR merging
Homework

• Today: WA2 Due
• Tuesday: Chapter 3.1 - 3.6
  - Optional Wikipedia Article
• Next Friday: PA3 due
  - Parsing!
• Tuesday Feb 27 - Midterm 1 in Class