LR Parsing, LALR Parser Generators

Lecture 10
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
  \[ \alpha \Rightarrow \gamma \]
  - \( \alpha \) is a stack of terminals and non-terminals
  - \( \gamma \) is the string of terminals not yet examined

- Initially: \( \Rightarrow x_1x_2 \ldots x_n \)
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

    \[
    E + (\triangleright \text{int} \triangleright) \Rightarrow E + (\text{int} \triangleright)
    \]

  • **Reduce** pops 0 or more symbols off the stack (the rule’s rhs) and pushes a non-terminal on the stack (the rule’s 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 $tok$ after
  - 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

0: int
1: E → int
2: int + (int) + (int)$ shift
3: int + (int) + (int)$ E → int
4: E + (int)$ shift(x3)
5: E + (E)$ shift(x3)
6: E + (E)$ E → int
7: E → E + (E)
8: E + (E)$ shift
9: E + (E)$ E → E+(E)
10: E → E + (E)
11: E → E + (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></td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
<td>g6</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→int</td>
<td>r_E→int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r_E→E+(E)</td>
<td></td>
<td></td>
<td></td>
<td>r_E→E+(E)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- sk is shift and goto state k
- r_X→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

• Remember for each stack element on which state it brings the DFA; use extra memory.

• 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
The LR Parsing Algorithm

Let $I = w$ be initial input
Let $j = 0$
Let DFA state 0 be the start state
Let stack = $\langle$ 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 $\langle X, Goto[top_state(stack), X]\rangle$
        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 \texttt{int} or a $E + (E)$ rhs
LR(1) Items

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

• \([X \rightarrow \alpha \cdot \beta, a]\) 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 $\triangleright$ was used before to separate the stack from the rest of input
  - $\alpha \triangleright \gamma$, where $\alpha$ is the stack and $\gamma$ is the remaining string of terminals

• In items $\bullet$ is used to mark a prefix of a production rhs:
  \[ X \rightarrow \alpha \bullet \beta, \alpha \]
  - Here $\beta$ might contain 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

• The initial parsing context contains:
  \[ S \rightarrow \cdot E, \$
  - Trying to find an $S$ as a string derived from $E\$
  - The stack is empty
LR(1) Items (Cont.)

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

• Consider the item
  \[ E \rightarrow E + ( \cdot E ), + \]

• 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:
  \[ E \rightarrow \cdot \text{int}, ) \]
  \[ E \rightarrow \cdot E + ( E ), ) \]
The Closure Operation

• Extending the context with items is called the closure operation.

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

• Construct the start context: Closure({$S \rightarrow \bullet E, \$})

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

• We abbreviate as:

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

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

• The start state contains \( [S \rightarrow \bullet E, \$] \)

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

• And now the transitions ...
The DFA Transitions

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

Transition(State, y)

\[
\begin{align*}
\text{Items} & \leftarrow \emptyset \\
\text{for each } [X \rightarrow \alpha \cdot y \beta, b] & \in \text{State} \\
\text{add } [X \rightarrow \alpha y \cdot \beta, b] & \text{ to Items} \\
\text{return Closure(Items)}
\end{align*}
\]
Constructing the Parsing DFA. Example.

```
S → •E, $
E → •E+(E), $/+ 
E → •int, $/+ 

S → E•, $
E → E•+(E), $/+ 

E → E+(E•), $/+ 
E → E•+(E), )/+ 

E → int•, $/+ 
E → int, $/+ 
E → E+(•E), $/+ 
E → •E+(E), )/+ 
E → •int, )/+ 

E → int•, )/+ 
E → int, )/+ 
E → int, $/+ 
E → int on $, + 
E → int on $, + 
E → int on $, + 
E → int on $, + 
E → int on $, + 
```

and so on...
LR Parsing Tables. Notes

• Parsing tables (i.e. the DFA) can be constructed automatically for a CFG

• Why study this at all in CS164? 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
  \([X \to \alpha\bullet a\beta, b] \text{ and } [Y \to \gamma\bullet, a]\)

• Then on input “a” we could either
  - Shift into state \([X \to \alpha a\bullet\beta, b], \text{ or}
  - Reduce with \(Y \to \gamma\)

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

- They are a typical symptom if there is an ambiguity 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
  \[
  [S \rightarrow \text{if } E \text{ then } S, \bullet, \text{ else}],
  [S \rightarrow \text{if } E \text{ then } S \bullet \text{ else } S, \ x]
  \]
- If *else* follows then we can shift or reduce
- Default (bison, CUP, JLALR, etc.) is to shift
  - Default behavior is right in this case
More Shift/Reduce Conflicts

• Consider the ambiguous grammar

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

• We will have the states containing

\[
\begin{align*}
[E \rightarrow E \ast \bullet E, +] & \quad \text{or} \quad [E \rightarrow E \ast E \bullet, +] \\
[E \rightarrow \bullet E + E, +] & \quad \Rightarrow^E \quad [E \rightarrow E \bullet + E, +]
\end{align*}
\]

\[ \ldots \quad \Rightarrow^{E} \quad \ldots \]

• Again we have a shift/reduce on input +
  - We need to reduce (\(* \) binds more tightly than +)
  - Solution: somehow impose the precedence of * and +
More Shift/Reduce Conflicts

Some parser generators (YACC, BISON) provide precedence declarations.
- Precedence left PLUS,
- Precedence left TIMES
- Precedence right EXP

• Bison, YACC
  - Declare precedence and associativity:
    \%left +
    \%left *

Prof. Fateman  CS 164  Lecture 10
More Shift/Reduce Conflicts

Our LALR generator doesn’t do this. Instead, we “Stratify” the grammar. (Less explanation!)

\[
E \rightarrow E + E \mid E * E \mid \text{int} \quad \text{;; original}
\]

New

\[
E \rightarrow E + E1 \mid E1
\]

\[
E1 \rightarrow E1 * \text{int} \mid \text{int}
\]

(Many “layers” may be necessary for elaborate languages. (13 in C++, and some operators appear at several levels, e.g. “(“. Some operators are right-associative like =, +=; most are left associative.)
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
  $$S \rightarrow \varepsilon \, | \, id \, | \, id \, S$$

• There are two parse trees for the string id
  $$S \rightarrow id$$
  $$S \rightarrow id \, S \rightarrow id$$

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

• Consider the states

  \[ S' \to \bullet \ S, \ $ \]  \[ S \to \text{id} \ \bullet, \ $ \]
  \[ S \to \bullet, \ $ \]  \[ S \to \bullet \ S, \ $ \]
  \[ S \to \bullet \ \text{id}, \ $ \]  \[ S \to \bullet \ \text{id}, \ $ \]
  \[ S \to \bullet \ \text{id} \ S, \ $ \]  \[ S \to \bullet \ \text{id} \ S, \ $ \]

  \[ \Rightarrow^{\text{id}} \]

• Reduce/reduce conflict on input $ $

  \[ S' \to S \to \text{id} \]
  \[ S' \to S \to \text{id} \ S \to \text{id} \]

• Better rewrite the grammar: \[ S \to \varepsilon \ | \ \text{id} \ S \]
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.

  $E \rightarrow \text{int} \bullet, $/+$$
  $E \rightarrow \text{int} \quad $on $, +$$

  and

  $E \rightarrow \text{int} \bullet, )/+$$
  $E \rightarrow \text{int} \quad $on ), +$$

- 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} \bullet, $/+/$$
  $E \rightarrow \text{int} \quad $on $, +, )$$
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
  \[ \{ [X \rightarrow \alpha \cdot \beta, b], [Y \rightarrow \gamma \cdot \delta, d] \} \]
is
  \[ \{ X \rightarrow \alpha \cdot \beta, Y \rightarrow \gamma \cdot \delta \} \]
LALR States

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

• They have the same core and can be merged
  and the merged state contains:
  \{[X \rightarrow \alpha, a/b], [Y \rightarrow \beta, 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

```
A -> B -> C
D -> E -> F
```

```
A --BE-- C
D --BE-- F
```
Conversion LR(1) to LALR(1). Example.
The LALR Parser Can Have Conflicts

• Consider for example the LR(1) states
  \{[X \to \alpha\bullet, a], [Y \to \beta\bullet, b]\}
  \{[X \to \alpha\bullet, b], [Y \to \beta\bullet, a]\}

• And the merged LALR(1) state
  \{[X \to \alpha\bullet, a/b], [Y \to \beta\bullet, 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

• You may see claims that any reasonable programming language has a LALR(1) grammar, {Arguably this is done by defining languages without an LALR(1) grammar as unreasonable 😊}. 

• In any case, LALR(1) has become a standard for programming languages and for parser generators, in spite of its apparent complexity.
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

• Next time we move on to semantic analysis
Notes on Lisp LALR generator

- `lalr.cl` is source code; `lalr.doc` additional documentation.
- A complete parse table can be viewed by
  - `(setf p (makeparser G lexforms nil))`
  - `(Print-Table stateList)`
  - `(eval p) ;; create the parser named LALR-parser`
Sample input for lalr generator

(defparameter G2 '(
    (exp --> exp + term  #'(lambda(exp n term)(list '+ exp term)))
    (exp --> exp - term  #'(lambda(exp n term)(list '- exp term)))
    (exp --> term         #'(lambda(term) term))
    (term --> term * factor #'(lambda(term n fac)(list '* term fac)))
    (term --> factor      #'(lambda(factor) factor))
    (factor --> id        #'(lambda(id) (const-value id)))
    (factor --> |(| exp |)| #'(lambda(pl exp p2) exp))
    (factor --> iconst    #'(lambda(iconst) (const-value iconst)))
    (factor --> bconst    #'(lambda(bconst) (const-value bconst)))
    (factor --> fconst    #'(lambda(fconst) (const-value fconst))))

(defparameter lexforms '( + - * |(| |)| id iconst bconst fconst))

(make-parser G2 lexforms nil)
Sample table-output for lisp lalr generator

STATE-0:
$Start --> . exp, nil
   On fconst shift STATE-14
   On bconst shift STATE-13
   On iconst shift STATE-12
   On ( shift STATE-7
   On id shift STATE-6
   On factor shift STATE-11
   On term shift STATE-16
   On exp shift STATE-1

STATE-1:
$Start --> exp ., nil
   exp --> exp . + term, + - nil
   exp --> exp . - term, + - nil
   On + shift STATE-9
   On - shift STATE-2
   On nil reduce exp --> $Start ... up to state 16
Each state is embodied in a subroutine

Defined locally in one main program via “labels”
Using local subroutines that shift, reduce, peek at next input
Main parser is called by (lalr-parser #'next-input #'error)

Any number of parsers can be set up in the same environment, though usually only one is tested... I just try out some input

(parse-fl '( (id a) + (id b)))

;; if there is a problem, edit the grammar, say G2, then

(remake G2)

(remakec G2) ;; COMPILES lalr-parser. Parser runs 20X faster or so.
Sample output program for lalr generator

(defun lalr-parser (next-input parse-error)
  (let ((cat-la 'nil) (val-la 'nil) (val-stack 'nil) (state-stack 'nil))
    (labels ((input-peek nil …;;these 3 subprograms are standard
      (shift-from (name) …
      (reduce-cat (name cat ndaughters action)…
      (STATE-0 nil  ;; generated specifically from grammar
        (case (input-peek)
          (fconst (shift-from #'STATE-0) (STATE-14))
          …
          (exp (shift-from #'STATE-0) (STATE-1))
          (otherwise (funcall parse-error)))))
      (STATE-1 nil
        (case (input-peek)
          (+ (shift-from #'STATE-1) (STATE-9))
          (- (shift-from #'STATE-1) (STATE-2))
          ((nil) (reduce-cat #'STATE-1 '$Start 1 nil))
          (otherwise (funcall parse-error)))
        …;etc etc)
Supplement to LR Parsing

Strange Reduce/Reduce Conflicts Due to LALR Conversion
Strange Reduce/Reduce Conflicts

- Consider the grammar

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

- \( P \) - 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

\[ P \rightarrow \bullet T \quad id \]
\[ P \rightarrow \bullet NL : T \quad id \]
\[ NL \rightarrow \bullet N \quad : \]
\[ NL \rightarrow \bullet N \quad , \quad NL \quad : \]
\[ N \rightarrow \bullet id \quad : \]
\[ N \rightarrow \bullet id \quad , \]
\[ T \rightarrow \bullet id \quad id \]
\[ T \rightarrow \bullet id \quad , \]
\[ N \rightarrow \bullet id \quad : \]

1. LALR merge
2. LALR merge
3. LALR reduce/reduce conflict on “,”
4.
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