LR Parsing pt 1: Bottom-Up and Shift-Reduce Parsing

CS 440: Programming Languages and Translators, Spring 2020

3/20 pp.4,7,8, 3/21, pp.3,5,6

1. Bottom-Up Parsing

- In bottom-up parsing, we build the smallest subtrees at the outskirts of the parse tree before combining them to build their parents, which we in turn combine to form those parents, and so on, up to the root of the parse tree. By comparison, with top-down predictive parsing, we have to know which rule to apply to the current nonterminal based on one symbol of lookahead (in LL(1)). Bottom-up parsing doesn't have that limitation.
- In bottom-up parsing, we don't always know exactly which rule we're parsing.
  - If we're in the right-hand side of either rule $A \rightarrow$ `x y z` or $B \rightarrow$ `x y w`, if we've only seen `x` and `y`, we won't know which rule we're parsing until we see either `z` or `w` as the next character.
  - More generally, if we're in the right-hand side of $A \rightarrow \alpha$ or $B \rightarrow \beta$ where $\alpha \rightarrow^* w\ldots$ and $\beta \rightarrow^* w\ldots$ (i.e., they can generate terminal strings with the initial substring), then we can't know which rule we're in until we get to the end of that initial substring.
  - (This leads to all sorts of questions like "What if $\alpha = \beta$?" "What if α and $\beta \rightarrow^*$ the same terminal string?" "What if $\alpha \rightarrow^* w$ and $\beta \rightarrow^* w\ldots$?" (The yield of α is an initial substring of the yield of β.)
- A bottom-up parser reads its input from left-to-right and builds small parse trees starting from the bottom left of the eventual overall parse tree, combing them to form larger bottom-left parse trees.
- A top-down LL parse visits the parse tree nodes in preorder. A bottom-up parse visits the nodes in postorder and generates a reverse rightmost derivation.
- **Example**: With the grammar $S \rightarrow T\ U\ T$, $T \rightarrow$ `a b`, $U \rightarrow$ `c c`, we can derive `a b c c a b` using rightmost derivations with:

  $S \rightarrow T\ U\ T \rightarrow T\ U$ `a b` $\rightarrow T$ `c c a b` $\rightarrow$ `a b c c a b`

  The reverse of the derivation is

  `a b c c a b` $\leftarrow T$ `c c a b` $\leftarrow T\ U$ `a b` $\leftarrow T\ U\ T \leftarrow S$
- So as we read the input, once we see `a b`, we can replace that by T, then we read `c c` and replace that with U, then we see the second `a b` and replace that T. The last transition, $T\ U\ T \leftarrow S$ is different: We don't read any extra input. When we go from $T\ U$ `a b` to $T\ U\ T$, we have to notice that $T\ U\ T \leftarrow S$ and do that replacement too. (In general, doing one replacement can lead to a cascade of successive replacements.)
  - This didn't happen with T `c c` $\ldots \leftarrow T\ U\ \ldots$ because there's no rule *nonterminal* $\rightarrow T\ U$.

2. Shift-Reduce Parsing

- Shift-reduce parsers are the kind of bottom-up parser normally seen when people talk about "bottom-up parsing".

- A **shift-reduce** parser uses the technique above with a stack to hold the symbols we've seen so far (a collection of terminals &/or parsed nonterminals) but haven't yet parsed completely. If a parsed nonterminal is represented as a parse tree, it's the full tree, not a partially-built tree.
- Shift-reduce parser reads the input left-to-right and maintains a stack of all the previously-seen partly-used parse trees and symbols.
  - E.g., with `x + y`, at the point where we've seen `x` and `+`, we have two items at the top of the stack
    - If we've parsed *Thing* → `x`, then we have *Thing* and `+`. If not, we have `x` and `+` on the stack
    - Items on the stack have been parsed as much as possible; if *Thing* → `x`, we don't have `x` and `+`..
- The shift-reduce name comes from the two basic operations of the parser.
  - **Shift** - the lookahead symbol (i.e., the next input symbol) is pushed onto the stack of unprocessed items.
  - **Reduce** - Remove the *n* most recent items on the stack, use it as the rhs of a rule (with *n* symbols in its rhs), and add the lhs nonterminal to the stack.
    - If that causes the top elements of the stack to look like the rhs of a rule, then reduce that too; repeat until no more reductions can be done.
    - This ensures that the stack only contains items that have been as completely parsed as possible.
    - This lets us look at only the roots of trees on the stack (if they're trees) to see that we have the rhs of some rule. (If we didn't reduce as much as possible, we'd have to look at something on the stack and realize we could reduce it.)
- Shift-reduce parsers differ according to how and when they decide to do a reduce step and how they handle possible conflicts between shifting / reduction decisions.

3. Example of Shift-Reduce Parsing

- Here's an example of shift-reduce parsing and how generates a reverse rightmost derivation.
- **Grammar:**
  - $S \rightarrow T\ U\ T$
  - $T \rightarrow$ `a b`
  - $U \rightarrow$ `c c`
- **Derivation**: $S \rightarrow T\ U\ T \rightarrow T\ U$ `a b` $\rightarrow T$ `c c a b` $\rightarrow$ `a b c c a b`
- We start at left end of the input a `b c c` a `b`. As we move through the input, we'll build a collection of terminal and nonterminal symbols. The terminal symbols are directly from the input; the nonterminals represent the roots of various parse trees that we've managed to parse so far.
- The character to the right of the _ is the next input character to look at. The things to the left of the _ are items that we haven't completely parsed yet. To shift, we move the _ right one symbol; to reduce using a rule $A \rightarrow \alpha$, we remove α from our list and replace it with A. Note α must be to the immediate left of the _. If α is just ε, we simply add A, to the left of the _.

_a b c c a b	See terminal a, add it to what we've seen = "Shift a".
a_b c c a b	Shift b
a b_c c a b	Reduce using $T \rightarrow$ a b (replace a b by T)
T_c c a b	Shift c
T c_c a b	Shift c
T c c_a b	Reduce using $U \rightarrow$ c c
T U_a b	Shift a
T U a_b	Shift b
T U a b_	Reduce using $T \rightarrow$ a b
T U T_	Reduce using $S \rightarrow T\ U\ T$
S_	Done!

- Going upward, we can read the derivation as $S \rightarrow T\ U\ T \rightarrow T\ U$ a b $\rightarrow T$ c c a b $\rightarrow$ a b c c a b, which is the rightmost derivation.

4. LR Parsing and LR(0) Items

Parts of an LR Parser

- An LR parser reads its input from Left to right and develops a reversed Rightmost derivation.
- It's a shift-reduce parser that uses a stack to hold the input that hasn't been completely processed. The stack can hold terminal symbols, of course, but it can include nonterminal symbols too because if we see the rhs of a rule like $B \rightarrow \beta$, we can use a reduction to replace the β on the stack by B.
- **Example**: Say we have rules $A \rightarrow$ a B e and $B \rightarrow$ b c d and input a b c d e. To process, we'll shift a, b, c, and d, onto the stack, reduce using b c d $\leftarrow B$ (so that the stack holds a B), then we'll shift the e, see that the stack holds a B e and reduce that to A.
- An LR parser is a kind of pushdown automaton, so it has a stack to hold data and a state machine to control what actions the parser takes.
- The automaton just has two actions: *shift* a terminal symbol on to the stack, or *reduce* the top of the stack using some rule. (The top of the stack contains the rhs of some rule, we pop it off the stack and push on the lhs of the rule we reduced by.)
- The automaton state keeps track of what rule we think we're parsing and where we are in the rule.
  - E.g., with $A \rightarrow$ a B e, we might be looking for an a, looking for a B, or looking for an e. Similarly with $B \rightarrow$ b c d, we might be looking for a b, c, or d. In addition, we can be at the end of a rule, which tells us we've seen the complete rhs of the rule and can reduce that to the lhs nonterminal.
- The state gets used
  - When we shift: Which rule(s) are we parsing now and where are we in their parses?
  - When we reduce: Which rule do we reduce with?
  - Just after we reduce: What state do we [3/21] go back to once we reduce?
  - **Example**: Say again we have rules $A \rightarrow$ a B e and $B \rightarrow$ b c d, and input a b c d e. When we shift the symbol a, we can start looking for a B: Once we see b c d and reduce that to B, we go back to the

$A \rightarrow$ a B e rule and now we're looking for an e. Note we might have multiple rules with B on the rhs, like $A \rightarrow$ a B e and $C \rightarrow B$ c, so this question isn't trivial.

LR(0) items as LR parsing states

- In the LR parsers we'll look at first, the states are **LR(0) items**, which are production rules decorated by adding a dot to indicate where we are in the rule. The dot starts at the left end, and whenever we see the symbol after the dot, we move it over to the right. Once the dot reaches the right end, we've seen the entire rhs and can reduce.
- **Example**: For $A \rightarrow$ a B e, the LR(0) items would be $A \rightarrow \bullet$ a B e, $A \rightarrow$ a $\bullet$ B e, $A \rightarrow$ a $B \bullet$ e, and $A \rightarrow$ a B e $\bullet$. If a rule has a rhs of length n, then we get n+1 items for that rule.
- For each state where the dot isn't at the end, we have a transition involving the symbol to the right of the dot. (E.g., state $A \rightarrow \bullet$ a B e goes to state $A \rightarrow$ a $\bullet$ B e on input a.)
- In addition, for each state like $A \rightarrow$ a $\bullet$ B e, where the dot is to the left of a nonterminal, we add ε-transitions to all rules for that nonterminal. So $A \rightarrow$ a $\bullet$ B e has an ε jump to $B \rightarrow \bullet$ b c d, and if there were other rules for B, we'd also ε-jump there, nondeterministically.
  - Symbolically, every state of the form $X \rightarrow \alpha \bullet Y \gamma$ has ε-jumps to all $Y \rightarrow \bullet \delta$ states.
- When we reach the end of the rule, $B \rightarrow$ b c d $\bullet$, we can reduce (pop b c d off the stack and push on B). When we starting parsing the B rule, we had just jumped there from $A \rightarrow$ a $\bullet$ B e, so once we reduce and push B onto the stack, we jump back to the A rule. Since we've seen the B, we jump to $A \rightarrow$ a $B \bullet$ e.
  - Symbolically, if we reduce $Y \rightarrow \delta \bullet$, then we jump back to the X rule we came from, to $X \rightarrow \alpha\ Y \bullet \gamma$.
  - This means as part of jumping to $Y \rightarrow \bullet \delta$, we have to remember that we jumped from $X \rightarrow \alpha \bullet Y \gamma$. And it's important to make that distinction, because we might have come from a different $lhs \rightarrow \ldots \bullet Y \ldots$ rule.
  - To keep that information, our stack will not only contain terminal and nonterminal symbols, it will contain the state we were in when we saw a symbol. So the stack will include information saying we were in state $X \rightarrow \alpha \bullet Y \gamma$ before we went to $B \rightarrow \bullet \delta$.
- Using nondeterministic LR parsers isn't practical, but we can use the set-of-states transform to get an equivalent deterministic machine.

Example: A small nondeterministic LR parser [3/20]

- For our example, take the grammar $S' \rightarrow S$ \$, $S \rightarrow$ a A, $A \rightarrow$ c and parse the input a c \$. Note the reductions below are in reverse rightmost derivation order: a c \$ $\leftarrow$ a A \$ $\leftarrow S$ \$ $\leftarrow S'$.

Stack	Input	Action	State
0	a c $	Shift a	State 0: $\{S' \to \bullet\ S\ \$,\ S \to \bullet\ \mathtt{a}\ A\}$
0 a 1	c $	Shift c	State 1: $\{S \to \mathtt{a} \bullet A,\ A \to \bullet\ \mathtt{c}\}$
0 a 1 c 2	$	Reduce via $A \to \mathtt{c} \bullet$	State 2: $\{A \to \mathtt{c} \bullet\}$
0 a 1 A 3	$	Reduce via $S \to \mathtt{a}\ A \bullet$	State 3: $\{S \to \mathtt{a}\ A \bullet\}$
0 S 4	$	Shift $	State 4: $\{S' \to S \bullet \$\}$
0 S 4 $ 5	ε	Reduce via $S' \to S\ \$$	State 5: $\{S' \to S\ \$ \bullet\}$
0 S' 6	ε	Accept, since input is ε	State 6: Accept unique top-level nonterminal

A Larger Example

- Let's look at the grammar $S' \to S\ \$,\ S \to A\ \mathtt{a},\ S \to B\ \mathtt{b},\ A \to \mathtt{c},\ B \to \mathtt{c}$.
- There are 13 LR(0) items:
  - $S' \to S\ \$$ generates $S' \to \bullet\ S\ \$,\ S' \to S \bullet \$,\ S' \to S\ \$ \bullet$
  - $S \to A\ \mathtt{a}$ generates $S \to \bullet\ A\ \mathtt{a},\ S \to A \bullet \mathtt{a},\ S \to A\ \mathtt{a} \bullet$
  - $S \to B\ \mathtt{b}$ generates $S \to \bullet\ B\ \mathtt{b},\ S \to B \bullet \mathtt{b},\ S \to B\ \mathtt{b} \bullet$
  - $A \to \mathtt{c}$ generates $A \to \bullet\ \mathtt{c},\ A \to \mathtt{c} \bullet$
  - $B \to \mathtt{c}$ generates $B \to \bullet\ \mathtt{c}\ ,\ B \to \mathtt{c} \bullet$
- The items induce 7 transitions along the 4 chains
  - From $S' \to \bullet\ S\ \$$ (via S) to $S' \to S \bullet \$$ (via $) to $S' \to S\ \$ \bullet$
  - etc.
- In addition, there are ε jumps from
  - $S' \to \bullet\ S\ \$$ to $S \to \bullet\ A\ \mathtt{a}$ and $S \to \bullet\ B\ \mathtt{b}$
  - $S \to \bullet\ A\ \mathtt{a}$ to $A \to \bullet\ \mathtt{c}$
  - $S \to \bullet\ B\ \mathtt{b}$ to $B \to \bullet\ \mathtt{c}$
- The deterministic automaton has an initial state of $\{S' \to \bullet\ S\ \$,\ S \to \bullet\ A\ \mathtt{a},\ S \to \bullet\ B\ \mathtt{b},\ A \to \bullet\ \mathtt{c},\ B \to \bullet\ \mathtt{c}\}$, with transitions
  - $\{S' \to S \bullet \$\}$ on S
  - $\{S \to A \bullet \mathtt{a}\}$ on A
  - $\{\ S \to B \bullet \mathtt{b}\}$ on B
  - $\{A \to \mathtt{c} \bullet,\ B \to \mathtt{c} \bullet\}$ on c
- This last state $\{A \to \mathtt{c} \bullet,\ B \to \mathtt{c} \bullet\}$ has two items in it because from the initial state includes $A \to \bullet\ \mathtt{c}$ and $B \to \bullet\ \mathtt{c}$, and nondeterministically, we don't know which of the two we're in. Thus if we see a c, we can progress through both rules, but we still don't know which one we're in.

[3/21 Conflicts moved to Lecture 18]

Activity Problems for Lecture 17

Lecture 17: Bottom-Up and (theory of) Shift-Reduce Parsing

1. With bottom-up parsing:
   a. How do we build parse trees?
   b. What kind of derivation does a bottom-up parser generate?
   c. Why don't we always know what production rule we're trying to parse?
   d. When do we have to figure out which rule we're parsing? Do we get to use local information? Non-local information?
   e. How does bottom-up parsing differ from predictive top-down parsing with LL(1) grammars?
2. In shift-reduce parsing, [3/21 Moved conflict question to Lecture 18]
   a. How do we read the input?
   b. What does the stack hold?
   c. What does the shift operation do?
   d. What does the reduce operation do?

Lecture 17: (Mechanics of) Shift-Reduce Parsing

1. Take the following grammar (found something similar on the web)

   $S \rightarrow X\ X\ \$$
   $X \rightarrow Y\ Y$
   $Y \rightarrow \texttt{0}\ Y \mid \texttt{1}\ Y \mid \texttt{3}\ Y \mid \texttt{4}$

   Use shift-reduce parsing to parse `0 0 4 1 0 1 4 3 3 4 4`

2. What would happen with parsing if we replaced the $Y \rightarrow \texttt{4}$ rule with $Y \rightarrow \varepsilon$?

3. Is the original grammar LL(1)? What about the $Y \rightarrow \varepsilon$ version?

4. List the LR(0) items for the grammar in Problem 1. Combine them using the set-of-states transform and list the states (sets of LR(0)) items that results.

Solutions to Activity Problems for Lecture 17

Lecture 17: Bottom-Up and (theory of) Shift-Reduce Parsing

1. (Bottom-up parsing)
   a. We built small subtrees at the frontier of the parse and combine them to form larger subtrees, eventually reaching the root.
   b. We generate the reverse of a rightmost derivation.
   c. What we've seen may go with the start of the rhs of multiple rules. E.g., $A \rightarrow \alpha\ \beta$ and $B \rightarrow \alpha\ \gamma$ look the same up through α.
   d. We have to know what rule we're parsing when we get to the end of its rhs. We get to use information from the part of the parse tree we've already built, so the information can be nonlocal.
   e. In LL(1) parsing, we build the parse tree from the root downward, and at every (non-leaf) parse tree node, we decide which rule to use at the beginning of the rhs of the rule (hence "predictive").

2. (Shift-reduce parsing)
   a. We read the input left to right.
   b. The stack holds the symbols we've seen so far: terminal symbols and parse trees for nonterminal symbols.
   c. The shift operations takes the next input symbol (a terminal symbol) and pushes it onto the stack.
   d. The reduce operation recognizes that we've seen the complete rhs of a rule: It pops the rhs symbols off the stack and pushes the lhs nonterminal onto the stack. (E.g., to reduce with $A \rightarrow B\ \texttt{c}$, we pop $B\ \texttt{c}$ off the stack and push A on.)

   [3/21 Moved conflict conflict questions to Lecture 18]

Lecture 17: (Mechanics of) Shift-Reduce Parsing

1. (Trace a shift-reduce parse) Our grammar is

   $S \rightarrow X\ X\ \texttt{\$}$
   $X \rightarrow Y\ Y$
   $Y \rightarrow \texttt{0}\ Y \mid \texttt{1}\ Y \mid \texttt{3}\ Y \mid \texttt{4}$

Trace:

_ 0 0 4 1 0 1 4 3 3 4 4 $	Start
0 _ 0 4 1 0 1 4 3 3 4 4 $	Shift 0
0 0 _ 4 1 0 1 4 3 3 4 4 $	Shift 0
0 0 4 _ 1 0 1 4 3 3 4 4 $	Shift 4
0 0 *Y* _ 1 0 1 4 3 3 4 4 $	Reduce $Y \to 4$
0 *Y* _ 1 0 1 4 3 3 4 4 $	Reduce $Y \to 0\ Y$
Y _ 1 0 1 4 3 3 4 4 $	Reduce $Y \to 0\ Y$
Y 1 _ 0 1 4 3 3 4 4 $	Shift 1
Y 1 0 _ 1 4 3 3 4 4 $	Shift 0
Y 1 0 1 _ 4 3 3 4 4 $	Shift 1
Y 1 0 1 4 _ 3 3 4 4 $	Shift 4
Y 1 0 1 *Y* _ 3 3 4 4 $	Reduce $Y \to 4$
Y 1 0 *Y* _ 3 3 4 4 $	Reduce $Y \to 1\ Y$
Y 1 *Y* _ 3 3 4 4 $	Reduce $Y \to 0\ Y$
Y Y _ 3 3 4 4 $	Reduce $Y \to 1\ Y$
X _ 3 3 4 4 $	Reduce $X \to Y\ Y$
X 3 _ 3 4 4 $	Shift 3
X 3 3 _ 4 4 $	Shift 3
X 3 3 4 _ 4 $	Shift 4
X 3 3 *Y* _ 4 $	Reduce $Y \to 4$
X 3 *Y* _ 4 $	Reduce $Y \to 3\ Y$
X Y _ 4 $	Reduce $Y \to 3\ Y$
X Y 4 _ $	Shift 4
X Y Y _ $	Reduce $Y \to 4$
X X _ $	Reduce $X \to Y\ Y$
X X $ _	Shift $
S	Reduce $S \to X\ X\ \$$
	Done!

2. (Replace $Y \to 4$ with $Y \to \varepsilon$)

The grammar becomes ambiguous if we replace $Y \to 4$ by $Y \to \varepsilon$.

$S \to X\ X\ \$$ $\quad X \to Y\ Y$ $\quad Y \to 0\ Y \mid 1\ Y \mid 3\ Y \mid \varepsilon$

For example, say we want to parse 0 0 $. If we start a derivation with

$S \to X\ X\ \$ \to Y\ Y\ X\ \$ \to Y\ Y\ Y\ Y\ \$$

then to derive parse `0 0 $`, it's sufficient to reduce two of the *Y*'s to ε and use $Y \rightarrow$ `0` $Y \rightarrow$ `0` ε on the other two *Y*'s. This yields 6 possible different parse trees. There other examples, for example with *Y Y Y Y* `$`, we can have the first $Y \rightarrow^*$ `0 0` ε and the other three *Y*'s → ε.

3. (LL(1)?)

In the original grammar, First(*X*) = First(*Y*) = {`0`, `1`, `3`, `4`}. There's only one rule for $X \rightarrow \ldots$, so there's no problem with selecting the wrong rule. The four rules $Y \rightarrow \ldots$ have different First sets, so their prediction table entries don't overlap. For example, First($Y \rightarrow$ `0` *Y*) = {`0`], so the Predict(*Y*, `0`) table entry = {$Y \rightarrow$ `0` *Y*}, and this is the only entry in which $Y \rightarrow$ `0` *Y* appears.

If we replace $Y \rightarrow$ 4 with $Y \rightarrow$ ε, then First(*X*) = First(*Y*) = {`0`, `1`, `3`} (we drop the `4`). Follow(*X*) = Follow(*Y*) = {`0`, `1`, `3`, `$`}. Since (for example), `0` ∈ Follow(*Y*), we have to add include $Y \rightarrow$ ε in Predict(*Y*, `0`), so Predict(*Y*, `0`) = {$Y \rightarrow$ `0` *Y*, $Y \rightarrow$ ε} and the grammar is not LL(1). The other prediction table entries for *Y* also wind up with two elements each, so the grammar is non-LL(1) three different ways.

4. (LR(0) items and states)

For the grammar $S \rightarrow X\ X$ `$`, $X \rightarrow Y\ Y$, $Y \rightarrow$ `0` Y | `1` Y | `3` Y | `4`, the LR(0) items are

- $S \rightarrow \bullet\ X\ X$ `$`, $S \rightarrow X \bullet X$ `$`, $S \rightarrow X\ X \bullet$ `$`, $S \rightarrow X\ X$ `$` $\bullet$
- $X \rightarrow \bullet\ Y\ Y$, $X \rightarrow Y \bullet Y$, $X \rightarrow Y\ Y \bullet$
- $Y \rightarrow \bullet$ `0` Y, $Y \rightarrow$ `0` $\bullet\ Y$, $Y \rightarrow$ `0` $Y \bullet$
- $Y \rightarrow \bullet$ `1` Y, $Y \rightarrow$ `1` $\bullet\ Y$, $Y \rightarrow$ `1` $Y \bullet$
- $Y \rightarrow \bullet$ `3` Y, $Y \rightarrow$ `3` $\bullet\ Y$, $Y \rightarrow$ `3` $Y \bullet$
- $Y \rightarrow \bullet$ `4`, $Y \rightarrow$ `4` $\bullet$

